PARALLEL COMPUTING OF OVERSET GRIDS FOR AERODYNAMIC PROBLEMS WITH MOVING OBJECTS

By

NATHAN C. PREWITT

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1999

Happy is the man that findeth wisdom, and the man that getteth understanding.

Proverbs 3:13

ACKNOWLEDGEMENTS

> Where no counsel is, the people fall: but in the multitude of counselors there is safety.
>
> Proverbs 11:14

That said, I would like to thank my counselors. Dr. Wei Shyy has been an excellent advisor, always enthusiastic and willing to help in any way possible. Likewise, Dr. Davy Belk has always been encouraging and supportive of my work. I am glad to be able to count them both as friends, as well as mentors. I would also like to thank the rest of my committee, Dr. Bruce Carroll, Dr. Chen Chi Hsu, and Dr. B. C. Vemuri. They have all been my allies.

I also want to thank Dr. Milton, who was the head of the GERC when I started this endeavor, and Dr. Sforza, who is now head of the GERC, and whom I have gotten to know better through AIAA. And since I am mentioning the GERC, thanks go to Cathy and Judi. I must also thank Mr. Whitehead, who has been my manager and an excellent supporter (even if he is an Alabama fan), since I came to work for QuesTech and now CACI. And, let me not forget Dr. Darryl Thornton (it gets worse, he's an Ole Miss grad), who has been my department director for many years now.

I have been very fortunate to receive funding from AFOSR. Without this funding and the time it allowed me away from my other task duties, I am sure that I would not have been able to complete my degree. I would like to thank everyone that helped to obtain this funding including Mr. Jim Brock and Maj. Mark Lutton of the Air Force Seek Eagle Office, and Mr. Dave Uhrig.

To be nice and legal, support was provided in part by a grant of HPC time from the DoD HPC Distributed Center, U.S. Army Space and Strategic Defense Command,

SGI Origin 2000. Additional support was provided by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F499620-98-1-0092. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

I hope that my presentation clearly shows which part of this work is my own. The original development of the Beggar code, including the unique algorithms for grid assemly, was done by Dr. Davy Belk and Maj. Ray Maple. The original work done to parallelize the Beggar flow solver was done by Dr. Davy Belk and Mr. Doug Strasburg. I owe a large debt to these gentlemen. There are now two others that continue the development of Beggar. Dr. Ralph Noack has been a tremendous asset to our group and has been most helpful in my work. It is from Ralph that I learned of the latency issues associated with the Origin 2000 and Ralph is the author of the load balancing algorithm used to distribute the grids in order to load balance the flow solver. Dr. Magdi Rizk came to Eglin at the same time that I did and we have worked together since. Magdi has been the sole developer of the Beggar flow solver for several years and has made significant contributions in that area. Magdi has also been my teacher at the GERC and a great friend. I would like to thank my other friends and colleagues and hope they will accept this thank you en masse.

Most importantly, I would like to thank my family. My in-laws have been very helpful (during births and hurricanes, pouring the patio and eating at Hideaway's) and have always been willing to take the wife and kids when I really needed to study (sometimes for weeks). My parents have always been supportive in innumerable ways, from driving for hours and then sitting in the car for hours at honor band tryouts, to attending MSU football games just to see the halftime show, to buying a house for us when Tracye and I got married in the middle of college, to doing my laundry

and fixing the brakes on the car over the weekend so I could get back to school. My brother Stephen has been a lot of help to my parents the last few years. Taking care of them is something that I don't think I could do. Jay and Josh are too much like me at times; but they are welcomed diversions and are stress relievers (our dog, Dudy Noble Polk, also fits into this category). And Tracye is a wonderful person. My roommate in college once said, "You are very lucky. You've found someone that you love, who is also a friend." She puts up with a lot. I am no picnic. We spent a lot of time apart during this, and I always get depressed whenever they are gone for a long time. Some of her bulldog friends, that she talks to over the internet, asked "How do you live with someone that is so positive all of the time?" "Who Tracye?", I replied. "No..., it's great!"

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

PARALLEL COMPUTING OF OVERSET GRIDS FOR AERODYNAMIC PROBLEMS WITH MOVING OBJECTS

By

Nathan C. Prewitt

December 1999

Chairman: Dr. Wei Shyy
Major Department: Aerospace Engineering, Mechanics and Engineering Science

When a store is dropped from a military aircraft at high subsonic, transonic, or supersonic speeds, the aerodynamic forces and moments acting on the store can be sufficient to send the store back into contact with the aircraft. This can cause damage to the aircraft and endanger the life of the crew. Therefore, store separation analysis is used to certify the safety of any proposed drop. This analysis is often based on wind tunnel aerodynamic data or analogy with flight test data from similar configurations. Time accurate computational fluid dynamics (CFD) offers the option of calculating store separation trajectories from first principles.

In the Chimera grid scheme, a set of independent, overlapping, structured grids are used to decompose the domain of interest. This allows the use of efficient structured grid flow solvers and associated boundary conditions, and allows for grid motion without stretching or regridding. However, these advantages are gained in exchange for the requirement to establish communication links between the overlapping grids via a process referred to as "grid assembly."

The calculation of a moving body problem, such as a store separation trajectory calculation, using the Chimera grid scheme, requires that the grid assembly be performed each time that a grid is moved. Considering the facts that time accurate CFD calculations are computationally expensive and that the grids may be moved hundreds of times throughout a complete trajectory calculation, a single store trajectory calculation requires significant computational resources.

Parallel computing is used regularly to reduce the time required to get a CFD solution to steady state problems. However, relatively little work has been done to use parallel computing for time accurate, moving body problems. Thus, new techniques are presented for the parallel implementation of the assembly of overset, Chimera grids.

This work is based on the grid assembly function defined in the Beggar code, currently under development at Eglin Air Force Base, FL. This code is targeted at the store separation problem and automates the grid assembly problem to a large extent, using a polygonal mapping (PM) tree data structure to identify point/volume relationships.

A logical succession of incremental steps are presented in the parallel implementation of the grid assembly function. The parallel performance of each implementation is analyzed and equations are presented for estimating the parallel speedup. Each successive implementation attacks the weaknesses of the previous implementation in an effort to improve the parallel performance.

The first implementation achieves the solution of moving body problems on multiple processors with minimum code changes. The second implementation improves the parallel performance by hiding the execution time of the grid assembly function behind the execution time of the flow solver. The third implementation uses coarse grain data decomposition to reduce the execution time of the grid assembly function. The final implementation demonstrates the fine grain decomposition of the grid assembly through the fine grain decomposition of the hole cutting process. Shared

memory techniques are used in the final implementation and appropriate dynamic load balancing algorithms are presented.

CHAPTER 1
INTRODUCTION

The knowledge of forces and moments induced by the addition of stores to an aircraft is vital for safe carriage. Once a store is released, knowledge of the interference aerodynamics and the effects on the trajectory of the store is vital for the safety of the pilot and aircraft. Such aerodynamic data have traditionally been provided by wind tunnel testing or flight testing; however, these techniques can be very expensive and have limitations when simulating time accurate, moving body problems such as the ripple release depicted in figure 1.1. Computational fluid dynamics (CFD) provides a way to supplement wind tunnel and flight test data.

Figure 1.1: History of three store ripple release

Overview

The primary problem to be considered is store separation from fighter aircraft configurations. The goal is to compute store separation trajectories in a timely fashion

using CFD and parallel computing. Due to the geometric complexity of aircraft/store configurations and the requirement to handle moving body problems, the Chimera grid scheme [1] is being used. This approach uses a set of overlapping structured grids to decompose the domain of interest. The Chimera grid scheme offers several advantages: 1) the use of structured grids allows the use of efficient block structured grid flow solvers and the associated boundary conditions; 2) the generation of overlapping grids which best fit a particular component geometry eases the burden of structured grid generation; and 3) the use of interpolation for communication between overlapping grids allows grids to be moved relative to each other. However, the communication between overlapping grids must be reestablished whenever a grid is moved. This process of establishing communication between overlapping grids will be referred to as *grid assembly.*

Whenever the grid around a physical object overlaps another grid, there is the probability that some grid points will lie inside the physical object and thus will be outside of the flow field. Even if no actual grid points lie inside the physical object, if a grid line crosses the physical object, there will be neighboring grid points that lie on opposite sides of the physical object. Any numerical stencil that uses two such neighboring grid points will introduce errors into the solution. This situation is avoided by cutting holes into any grids overlapping the physical surfaces of the geometry.

During hole cutting, regions of the overlapping grids are marked as invalid. This creates additional boundaries within the grid system. The flow solver requires that some boundary condition be supplied at these boundaries. Likewise, some boundary condition is also needed at the outer boundaries of embedded grids. Collectively, the grid points on the fringe of the holes and the grid points on the outer boundaries of the embedded grids are referred to as inter-grid boundary points (IGBPs) [2]. The boundary conditions required at the IGBPs are supplied by interpolating the flow solution from any overlapping grids.

The Beggar code [3], developed at Eglin Air Force Base, is capable of soving three-dimensional inviscid and viscous flow ploblems involving multiple moving objects, and is suitable for simulating store separation. This code allows blocked, patched, and overlapping structured grids in a framework that includes grid assembly, flow solution, force and moment calculation, and the integration of the rigid body, six degrees of freedom (6DOF) equations of motion. All block-to-block connections, patched connections, freestream boundary conditions, singularity boundary conditions, and overlapped boundaries are detected automatically. All holes are defined using the solid boundaries as cutting surfaces and all required interpolation stencils are calculated automatically. The integration of all necessary functions simplifies the simulation of moving body problems [4]; while the automation and efficient implementation of the grid assembly process [5] significantly reduces the amount of user input and is of great benefit in a production work environment.

The basic solution process consists of an iterative loop through the four functions shown in figure 1.2. The blocked and overset grid system is first assembled. Once this is done, the flow solution is calculated in a time-accurate manner. Aerodynamic forces and moments are then integrated over the grid surfaces representing the physical surfaces of the moving bodies. The rigid body equations of motion are then integrated with respect to time to determine the new position of the grids considering all aerodynamic forces and moments, forces due to gravity, and all externally applied forces and moments (such as ejectors).

Multiple iterations of this loop are required to perform a complete store separation trajectory calculation. The accuracy of the trajectory predicted from the itegration of the equations of motion is affected by the time step chosen; however, stability contraints on the flow solver are normally more restrictive. In typical store separation calculations, the time step has been limited to 0.1-1.0 milli-second; thus, hundreds or even thousands of iterations are often required.

As the complexity of flow simulations continues to increase it becomes more crit-

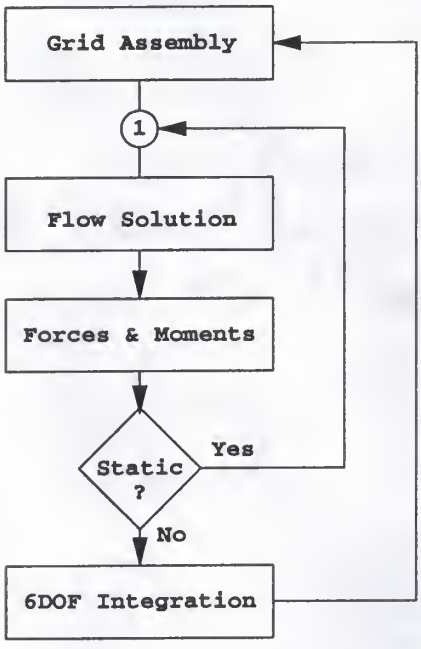


Figure 1.2: Solution process

ical to utilize parallel computing to reduce solution turnaround times. The parallel implementation of the Beggar flow solver was first presented by Belk and Strasburg [6]. This flow solver uses a finite volume discretization and flux difference splitting based on Roe's approximate Riemann solver [7]. The solution method is a Newton-Relaxation scheme [8]; that is, the discretized, linearized, governing equations are written in the form of Newton's method and each step of the Newton's method is solved using symmetric Gauss-Seidel (SGS) iteration. The SGS iterations, or *inner iterations*, are performed on a grid by grid basis; while the Newton iterations, or *dt iterations*, are used to achieve time accuracy and are performed on all grids in sequence. In this reference, the separate grids are used as the basis for data decomposition. The grids, which represent separate flow solution tasks, are distributed across multiple processors and the flow solver is executed concurrently. The only communication between processes is the exchange of flow field information at block-to-block, patched, and overlapped boundaries between dt iterations. The grid assembly is

performed only once; thus, only static grid problems are addressed.

It is also desirable to utilize parallel computing to reduce the turnaround time of moving body problems such as the ripple release configuration. In order to do so, the grid assembly function must be executed each time grids are moved. An efficient, scalable parallel implementation of any process requires that both the computation and the required data be evenly distributed across the available processors while minimizing the communication between processors. The movement of the grids and the time variation in the holes being cut, as illustrated in figure 1.3, indicate the dynamic and unstructured nature of the grid assembly work load and data structures. This makes an efficient implementation a challenging task.

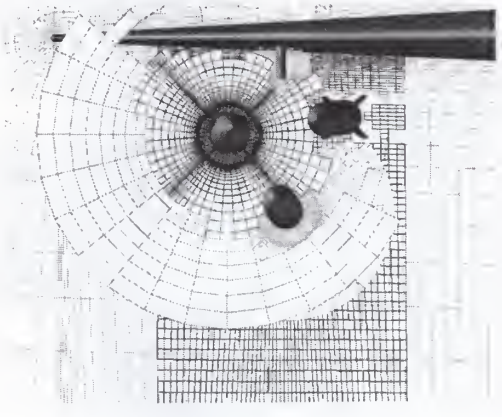


Figure 1.3: Example of overlapping grids with holes cut

Thus, the primary focus of this work is the parallel implementation of the grid assembly function so that store separation trajectories can be calculated using time-accurate CFD and parallel computers. A logical succession of incremental steps is used to facilitate the parallel implementation of the grid assembly function. The initial implementation (phase I) uses a single process to perform the entire grid assembly in a serial fashion with respect to the parallel execution of the flow solver. This requires that proper communication be established between the flow solution function and the grid assembly fuction; however, it does not require any consideration of load balancing or partitioning of the grid assembly function. The grid assembly function

is not accelerated, but the flow solution is.

In the second implementation (phase II), parallel efficiency is gained by overlapping the grid assembly function and the flow solution function. This overlapping of work is possible because of the use of the Newton-Relaxation method within the flow solver. Each step of the approximate Newton's method produces an approximation to the flow solution at the next time step. Approximate aerodynamic forces and moments are calculated from the flow solution after the first Newton step and are used to drive the grid assembly function, while additional Newton steps are being calculated to achieve time accuracy.

As long as there is sufficient time to hide the work of the grid assembly function, the speedup is affected only by the performance of the flow solver. However, as the processor count increases, the time of the flow solution available to hide the grid assembly decreases and the rate of change of speedup with respect to processor count decreases. Therefore, it is important to distribute the work of the grid assembly function to make the entire process scalable to higher processor counts.

The third implementation (phase III) uses data decomposition of the grid assembly function to reduce its execution time and thus allows the grid assembly time to be more easily hidden by the flow solution time. The basis for the data decomposition is the *superblock*, which is a group of grids that contain block-to-block connections and are overlapped with other superblocks. In this implementation, the work and data structures associated with a superblock are distributed over multiple processors. Dynamic load balancing is used to improve the performance by moving superblocks between processes.

The relatively small number of superblocks used in most problems places a limit on the number of processors that can be effectively utilized. Thus, in order to improve scalability, the fourth implementation (phase IV) uses a fine grain decomposition of the work associated with grid assembly. The work of the grid assembly function can be associated with the facets that cut holes into overlapping grids and the cell centers

that require interpolation. Therefore, the hole cutting facets and the IGBPs form the basis for the fine grain distribution of the work associated with grid assembly.

This dissertation gives complete details of the implementation options for including the grid assembly function into the parallel execution of moving body CFD computations. Each implementation builds upon the previous implementation, attacking the limitations in order to improve performance. Details of the performance analysis are included. Equations for estimating the performance are also presented. With practical experience and some further development, these implementations and performance estimators could offer optimum execution guidelines for particular problems.

Related Work

Grid Assembly

Table 1.1 lists some of the codes that are currently available for assembling overset grid systems. Some of the advantages and disadvantages of each code are listed. Since the author is not completely familiar with the operation of all of these codes, some of the disadvantages (or advantages) may only be perceived. In general, finding the root cause of a failure in the grid assembly process is a difficult task. Therefore, it is a disadvantage of overset grids in general and is not listed as a disadvantage for any of the codes although some of the codes provide better aids for debugging than do others. Likewise, the use of orphan points (points that fail to be properly interpolated and are given averaged values from neighbors) can help to ensure that grid assembly does not fail. However, orphan points are not listed as an advantage for any code since they can adversely affect the flow solution.

PEGSUS [9] is the first and one of the more widely used codes for handling the grid assembly problem. It relies on a set of overlapping grids (block-to-block connections are not allowed). PEGSUS is completely separate from any flow solver but will produce interpolation information for either finite difference or finite volume

Table 1.1: Grid assembly codes

Code	Advantage	Disadvantage
PEGSUS	First code; large user base	Slow; requires alot of user input
DCF3D	Fast; large user base; well supported	Requires significant user input
CMPGRD	Modern programming techniques; well defined algorithms	Not widely distributed
BEGGAR	Automated grid assembly; allows block-to-block connections; small user input geared toward production work environment; complete flow solution environment	Slower than DCF3D; monolithic code; limited user base; has difficulties with overset viscous grids

flow solvers. The amount of user input required is often rather large: each hole cutting surface has to be identified, all overlapping boundaries must be identified, and a set of links must be specified to tell the code which grids to cut holes into and where to check for interpolation coefficients.

DCF3D (domain connectivity function) [2] is another code used to accomplish the grid assembly task. DCF3D is not coupled directly with any flow solver but it has been used extensively with the OVERFLOW flow solver [10]. DCF3D uses several alternative approaches in order to improve the efficiency of the grid assembly process. DCF3D uses analytic shapes for hole cutting which allows grid points to be compared directly to the hole cutting surface. It also uses Cartesian grids, called inverse maps, to improve the efficiency of selecting starting points for the search for interpolation stencils. These techniques improve the efficiency of the grid assembly process; however, an additional burden is placed on the user to define the analytical shapes and the extent and density of the inverse maps.

More recently, improvements to DCF3D have been proposed in order to reduce the burden placed on the user. These improvements include the use of *hole-map* technology and the iterative adjustment of the connectivity information [11]. Hole-map

technology uses Cartesian grids to map the hole cutting surfaces in an approximate, stair stepped fashion. This would allow the automatic creation of the hole cutting surfaces and an efficient means of identifying hole points. The iterative process of adjusting the connectivity information by expanding and contracting the holes in order to minimize the overlap between grids also offers benefits.

Yet another code that addresses the grid assembly problem is CMPGRD [12]. This code is an early version of the grid assembly process that has been included in OVERTURE [13]. This tool does not appear to be highly optimized; moreover, its strengths seem to be in its well defined algorithms for the grid assembly process. The algorithms can produce minimum overlap between grids and other quality measures are considered in the donor selection process.

In comparison to the above mentioned codes, Beggar is unique in that its development has been geared towards the store separation problem and a production work environment. As such, Beggar attempts to automate the entire solution process while reducing the burden of input that is placed on the user. Beggar also uses unique data structures and algorithms in order to maintain the efficiency of the grid assembly process.

Store Separation

Table 1.2 lists some of the techniques that have been used to calculate store separation trajectories. Some of the advantages and disadvantages from each technique are listed. The techniques range from simple curve fits of data from similar configurations, to wind tunnel experimental methods, to the calculation of the complete flow field from first principles.

Engineering level methods (see Dillenius et al. [14] for example) derive aerodynamic data from data bases of experimental data, simple aerodynamic correlations, and panel methods with corrections for nonlinear effects such as vortical flow. Such methods are computationally inexpensive but have very limited applicability. These

Table 1.2: Store separation modeling methods

Method	Advantage	Disadvantage
Engineering Level Methods	Computationally inexpensive; provide quick data for preliminary design	Limited applicability
Captive Trajectory Support	Wind tunnel accuracy of flow phenomenon	Limited range of motion; quasi-steady; high cost; tunnel interference
Influence Function Method	Fast solution allows statistical investigation of trajectories	Mutual interference effects can be lost
Computational Fluid Dynamics	Completely time accurate; flexible; unlimited in configuration; provides data for visualization of the complete flow field	Grid generation can be labor intensive; requires significant computing resources; weaknesses in modeling some flow phenomena such as turbulence

methods are most useful in preliminary design, but have been applied to the calculation of store separation trajectories.

Store separation events have been simulated in wind tunnels using the captive trajectory support (CTS) system [15]. This technique places a sting mounted store in the flow field of an aircraft wing and pylon. The store is repositioned according to the integration of measured aerodynamic loads and modeled ejector loads. Since the store can not be moved in real-time, an angle-of-attack correction is made based on the velocity of the moving store. This technique is quasi-steady and often limited in the range of motion due to the sting mechanism.

Store separation trajectories have also been calculated using wind tunnel data and an influence function method (IFM) [16]. This method uses wind tunnel data to define flow angularity near an aircraft wing and pylon. These data are used to apply a delta to the freestream forces and moments of the store assuming that the store does not affect the aircraft flow field. Another correction is made for mutual interference using wind tunnel data of the store in carriage position. Jordan [17] gave a detailed comparison of loads calculated from IFM and CFD versus loads measured

in the wind tunnel. IFM was shown to be inaccurate due to mutual interference that is not directly related to flow angle. The distance at which mutual interference becomes insignificant must also be well known.

Such semi-emperical techniques can also be used with CFD data replacing part or all of the wind tunnel data. In a recent special session at the AIAA Aerospace Sciences Meeting, most of the papers [18, 19, 20] presented used this technique. One paper [21] used quasi-steady CFD. Of the two time-accurate CFD simulations slated to be presented, one was withdrawn and the other was prevented from being presented due to the failure to get clearance for public release.

When considering time-accurate CFD calculations for moving body problems, the decomposition of the domain (grid type) has a significant impact on the solution process. Table 1.3 lists several of the different grid methods in use. Some of the advantages and disadvantages of each grid method are listed.

Cartesian grids (see [22] for example) have been used for moving body problems, but the treatment of boundary counditions can be complicated. Boundary conforming block structured grids have been used to calculate store separation trajectories [23]; however, the motion of a store within a block structured grid requires grid stretching and deformation. This places a limit on the motion before regridding is required due to errors introduced by grid skewness. Unstructured grids have also been applied to the store separation problem (see [24] for example). The flexibility of unstructured grid generation eases the grid generation burden but complicates the flow solver. Octree grids have also been used to ease the grid generation burden and allow adaptive grid refinement. SPLITFLOW [25] represents a compromise between these unstructured grid techniques. A prismatic grid is used near solid surfaces to simplify boundary conditions and an octree grid is used away from the solid surfaces and offers adaption and some organizational structure. Chimera grid methods are also a compromise and have been applied extensively to the store separation problem [4, 26, 27, 28, 29, 30]. They can be viewed as locally structured, but globally unstructured.

Table 1.3: Grid generation methods

Grid Type	Advantage	Disadvantage
Cartesian	Small memory requirements; fast flow solver	Difficult treatment of boundary conditions; poor viscous solution capabilities
Structured	General treatment of flow solver and boundary conditions	Restricted to simple geometries
Block Structured	Extension to complex geometries	Grid generation is time consuming; grid motion or adaption is difficult
Quad Tree	Easily adapted	Difficult treatment of boundary conditions; connectivity information required
Unstructured	Automatic grid generation; easily adapted	Larger memory requirements; slower flow solvers; connectivity information required; weak viscous solution capabilities
Chimera	Structured grid flow solvers and boundary conditions; eases grid generation burden; allows grid movement	connectivity (only at IGBPs) must be constructed separate from the grid generation process

Time accurate CFD has been validated for use in calculating store separation trajectories. Lijewski [28] presented the first complete system for calculating store separation trajectories. Lijewski [28] also presented the first use of a particular set of wind tunnel CTS data for store separation code validation. The configuration is a single, sting mounted, ogive-cylinder-ogive store under a generic pylon and wing. Grids for the generic store are shown in figure 1.4.

Data, first presented by Prewitt et al. [4], for the subsonic and supersonic trajectories of the single generic store are shown in figures 1.5 and 1.6. The CTS data are shown by the symbols and the time accurate CFD calculations are shown by the curves. These comparisons show excellent agreement between the wind tunnel data and time accurate CFD calculations for this test problem.

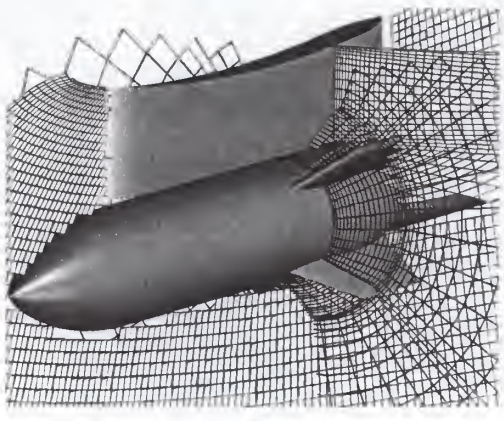


Figure 1.4: Grids for single generic store trajectory calculation

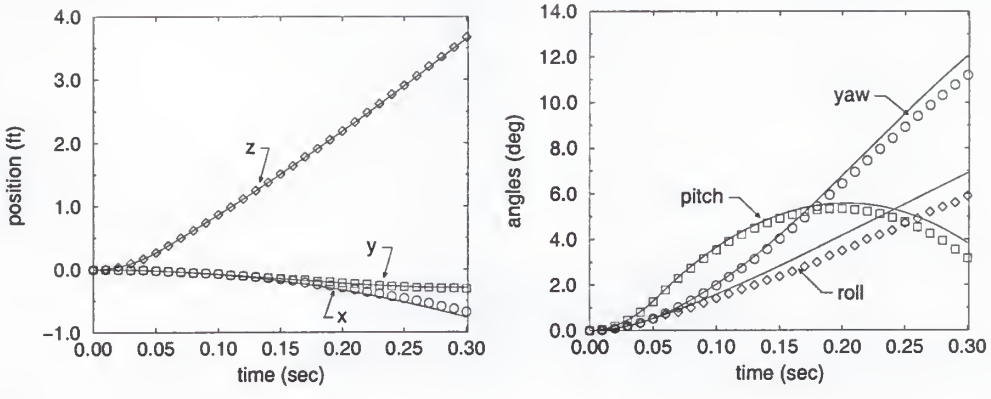


Figure 1.5: Mach 0.95 single store trajectory calculated (left) CG position and (right) angular position versus wind tunnel CTS data

More complex configurations have also been used for validation cases. Cline et al. [31] presented store separation trajectories from an F-16 aircraft configuration including a fuel tank, pylon, and an aerodynamic fairing at the junction of the pylon and the wing. Coleman et al. [32] presented separation trajectories for the MK-84 from the F-15E aircraft. This configuration included a centerline fuel tank, a LANTIRN targeting pod, an inboard conformal fuel tank (CFT) weapons pylon with attached MK-84, forward and middle stub pylons on the outside of the CFT, LAU-128 rail launchers with AIM-9 missiles on both sides of the wing weapons pylon, and the MK-84 to be released from the wing weapons pylon. Both references compared

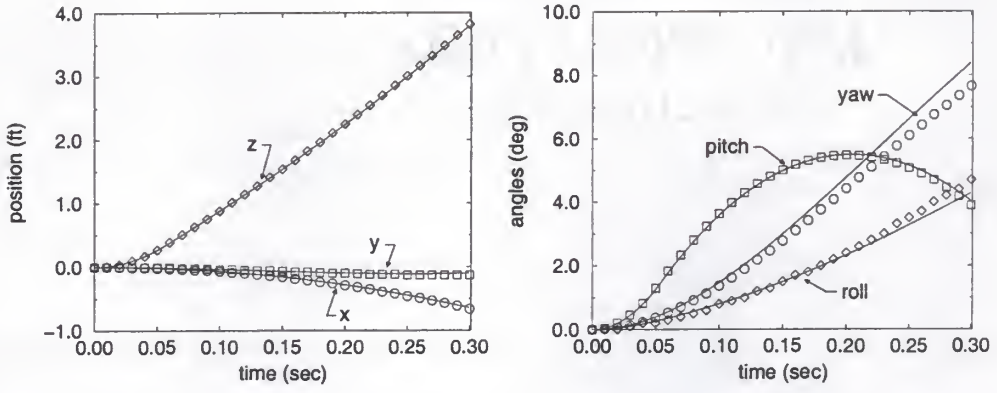


Figure 1.6: Mach 1.20 single store trajectory calculated (left) CG position and (right) angular position versus wind tunnel CTS data

trajectory calculations to wind tunnel data and were instrumental in the approval of the use of CFD by engineers in evaluating the store separation characteristics of weapons.

Parallel Computing

Although much work has been done on the parallel execution of CFD flow solvers, including Chimera method flow solvers, little work has been done on the efficient parallel implementation of Chimera methods for moving body problems. In particular, there are very few references on the parallel treatment of the grid assembly problem. Table 1.4 gives a list of references of some of the more important developments in parallel computing as related to Chimera grid methods and grid assembly.

Smith [33] presents the parallel implementation of an overset grid flow solver for a network based heterogeneous computing environment. This flow solver was derived from OVERFLOW and uses coarse grain parallelism with the component grids being distributed among the available processors. A master/slave model is used. The master process performs all i/o functions, maintains all of the interpolated flow solution data, and communicates with each of the slave processes. The slave processes calculate the flow solution and perform the interpolation of flow solution data. A load balancing

technique is used and the interpolation of flow solution data is overlapped with the calculation of the flow solution to reduce load imbalances. The grid communication information was supplied as an input and only static problems were addressed.

Wissink and Meakin [34] presented the application of a Chimera grid flow solver based on OVERFLOW and DCF3D. This code uses overlapping structured grids near the solid boundaries in order to resolve viscous effects and uses spatially refined Cartesian blocks in the rest of the domain. Parallel performance was presented but only static problems were addressed. The same code was again presented by Meakin and Wissink [35]. Two dynamic problems were presented in this reference; however, the focus was on the ability to adapt the Cartesian blocks due to flow solution and body motion. Some parallel performance data are presented based on an iteration of the flow solver. No performance data were presented for an entire simulation which would include the performance of the grid assembly.

The first presentation of the parallel implementation of grid assembly for dynamic, overset grids was by Barszcz et al. [36]. DCF3D was parallelized and used in connection with a parallel version of OVERFLOW on a distributed memory parallel machine. A coarse grain parallelism was implemented with the data decomposition based on component grids. A static load balance was used based on balancing the load of the flow solver. Since the flow solver represented a large portion of the total work, load balancing the flow solver is important to achieving a good overall load balance; however, significant imbalances were seen in the grid assembly processes. Donor cell identification was found to be the most time consuming part of grid assembly and algorithm changes were implemented to reduce this part of the work load.

Weeratunga and Chawla [37] again used DCF3D and OVERFLOW on a distributed memory parallel machine. Again, the component grids are used for data decomposition and load balancing is based on the work load of the flow solver. No consideration is given to the distribution of donor elements or IGBPs. The primary focus in this reference is on demonstrating the scalability of the processes used. In

this study, the donor search method scaled well; however, the hole cutting and the identification of points requiring interpolation did not scale well.

Wissink and Meakin [38] present the first attempt to load balance the grid assembly process. However, the data decomposition is still based on the component grids and affects the load balance of both the flow solver and the grid assembly function. A static load balance is initially performed to equally distribute the numbers of grid points which helps to load balance the flow solver. A dynamic load balancing routine is then used during a calculation to redistribute the grids to improve the load balance of grid assembly. This, in turn, creates an imbalance in the flow solver. This algorithm offers a method of improving performance if an imbalance in the grid assembly work load is a major deterrent. However, in the problems presented, the flow solver represented the major part of the work load and any redistribution of grids in order to improve the grid assembly load balance actually decreased the overall code performance.

Dissertation Outline

Chapter 2 presents details of the algorithms and data structures of the grid assembly process. For completeness, chapter 3 presents the flow solution algorithm and chapter 4 presents the integration of the 6DOF rigid body equations of motion. Chapter 5 presents an overview of programming parallel computers and outlines the approaches used in the current work. Chapter 6 gives some details of the proposed implementations including equations for estimating speedup. Chapter 7 presents the ripple release test problem used for all timings of the implementations. The results of the timings are presented in chapter 8. The final conclusions and some possibilities for future work are presented in chapter 9.

Table 1.4: Significant accomplishments in parallel computing in relation to overset grid methods

Reference	Accomplishment	Limitation
Smith and Pallis, 1993 [33]	Parallel implementation of OVERFLOW for heterogeneous computing environments	Restricted to static problems
Barszcz, Weeratunga, and Meakin, 1993 [36]	First parallel implementation of grid assembly	Data decomposition and static load balance tied to flow solver
Weeratunga and Chawla, 1995 [37]	Detailed study of scalability of parallel implementation of DCF3D	Data decomposition and static load balance tied to flow solver
Belk and Strasburg, 1996 [6]	First parallel implementation of Beggar	Restricted to static problems
Wissink and Meakin, 1997 [38]	First attempt to load balance grid assembly	Decomposition of grid assembly tied to flow solver means any improvement in the load balance of grid assembly adversely affects the load balance of the flow solver
Wissink and Meakin, 1998 [34]	Small, near body, curvilinear grids used in combination with adaptive Cartesian grids	Only static problems were presented
Prewitt, Belk, and Shyy, 1998 [39]	First parallel implementation of Beggar for dynamic problems; overlapping of grid assembly and flow solution time	Limited scalability
Meakin and Wissink, 1999 [35]	Included dynamic problems with combined overset grids and adaptive Cartesian grids	No performance of dynamic grid assembly was presented
Prewitt, Belk, and Shyy, 1999 [40]	Coarse grain decomposition and dynamic load balancing of grid assembly based on superblocks independent of flow solver	Major functions within the grid assembly are not individually well balanced

CHAPTER 2
GRID ASSEMBLY

Although Beggar is useful for general external compressible fluid flow problems, its primary focus during development has been on the simulation of store carriage and separation events. A typical grid system includes grids for an aircraft, pylons, launchers, and stores. The grids are often placed inside a large rectangular grid which serves as a background grid that reaches to freestream. Due to disparity in grid spacing between overlapping grids, it is often necessary to introduce other grids that serve as an interface to aid communication. The stores are often bodies of revolution with attached wings, canards, and/or fins. Blocked grid systems are used for these relatively simple geometries; however, in order to allow grid movement, such blocked grids are treated as overlapping grids with respect to other grids.

The *superblock* construct is introduced to aid in grid assembly. The superblock is a collection of non-overlapping grids which are treated as a single entity. Block-to-block connections are allowed only within a superblock; thus, a superblock is often used to implement a blocked system of grids for part of the solution domain. Overlapping connections are allowed only between different superblocks.

A *dynamic group* is a collection of one or more superblocks that is treated as a single entity by the 6DOF. The dynamic group is used primarily to group grids which are part of the same moving body. There is always at least one dynamic group: the static dynamic group. This holds the static grids like the background grid, the aircraft grid, pylon grids, or store grids that do not move relative to the aircraft. Other dynamic groups are created for each store that will move relative to the static dynamic group. Since a dynamic group may contain one or more superblocks, each

moving body can be constructed from a system of blocked grids in a single superblock, a system of overlapping grids in multiple superblocks, or a combination of the two.

Polygonal Mapping Tree

In order to establish overlapped grid communications the following questions must be answered: does this point lie inside a grid, and if so, what is an appropriate interpolation stencil? These questions represent point-volume geometric relationships. In order to determine such relationships, Beggar uses a *polygonal mapping* (PM) tree, which is a combination of the *octree* and *binary space partioning* (BSP) tree data structures [41, 42].

An octree is a data structure in which a region of space is recursively subdivided into *octants*. Each *parent* octant is divided into eight *children* which can be further subdivided. This forms a hierarchy of *ancestor* and *descendant* octants. Each octant in the tree is termed a *node* with the beginning node (lowest level) being the *root node* and the most descendent nodes (highest levels) being the *leaf nodes*. Such a data structure allows a domain to be divided into 8^n subdomains using just n levels. Associated with each node are the Cartesian coordinates of the center of the octant. Which child octant a point lies in can be identified by comparing the coordinates of the point against the coordinates of the center of the parent octant. With such a data structure, a point can be identified as lying within a particular octant out of 8^n octants by using at most n comparisons (if the tree is perfectly balanced).

The BSP tree is a binary tree data structure in which each node of the tree is represented by a plane definition. Each node has two children representing the *in* and *out* sides of the plane. For a faceted representation of a surface, each facet defines a plane that is inserted into the BSP tree. While being inserted the facet may be clipped against existing planes in the BSP tree placing pieces of the same plane definition into different branches of the tree. Using a given point, the BSP tree is traversed by comparing the point against a plane definition at each level to determine

which branch to descend into. Once a leaf node is reached, the point is identified as being inside or outside of the faceted surface.

In theory, a BSP tree of the cell faces on the boundaries of a grid block could be used to determine whether a point is IN or OUT of that particular grid. However, due to the clipping process, the BSP tree can be prone to roundoff error. Likewise, the structure of the tree is dependent on the order in which facets are inserted and it is not guaranteed to be well balanced. If the tree were to become one-sided, a point may have to be compared against all or most of the facets on a surface to determine its relationship to that surface. Therefore, Beggar uses a combination of the octree and BSP tree data structures. The octree, which stays well balanced, is used to quickly narrow down the region of space in which a point lies. If a point lies in a leaf node that contains an overlapping boundary grid surface, it must be compared to a BSP tree that is stored in that leaf node to determine its relationship to that boundary surface and therefore its relationship to the grid itself.

The PM tree data structure is built by refining the octree in a local manner until no octant contains more than one grid boundary point from the same superblock. This produces a regular division of space that adapts to grid boundaries and grid point density. The boundary cell faces of the grids are then used to define facets which are inserted into BSP trees stored at the leaf nodes of the octree. Since each grid boundary point is normally shared by four cell faces and each octant contains only one grid boundary point, the BSP trees stored at the octree leaf nodes should be very shallow.

Once the basic data structure is complete, all of the octants of the leaf nodes are classified relative to the grid boundaries. Each octant is classified as inside or outside of each superblock or as a *boundary* octant. Then points can be classified efficiently relative to the superblocks. To do so, the octant in which the point lies is found. If the octant has been classified as IN or OUT, the point can be immediately classified as IN or OUT. However, if the point lies in a boundary octant, the point must be

compared against the BSP tree that is stored in that octant.

Figure 2.1 represents a quadtree (2d equivalent of an octree) for a 4 block O-grid around an airfoil. Only the grid points on the boundaries are used to define the level of refinement, so only the boundaries of the grid are shown. The grid boundaries are inserted into the leaf octants as BSP trees to form the PM tree. A portion of the PM tree that might result is shown in figure 2.2. The leaf octants are represented by squares, while the other nodes are represented by circles. The four branches at each node represent the four quadrants of an octant. The line segments shown in some of the leaf octants represent portions of the grid boundaries that would be placed in BSP trees. If a point being classified against the PM tree falls into one of these leaf octants, it must be compared against the facets to determine its relationship to the grid. The empty leaf octants that are drawn with solid lines are completely inside the grid, while the leaf octants that are drawn with dashed lines are completely outside the grid. Points that fall into either of these types of octants can immediately be classified relative to the grid.

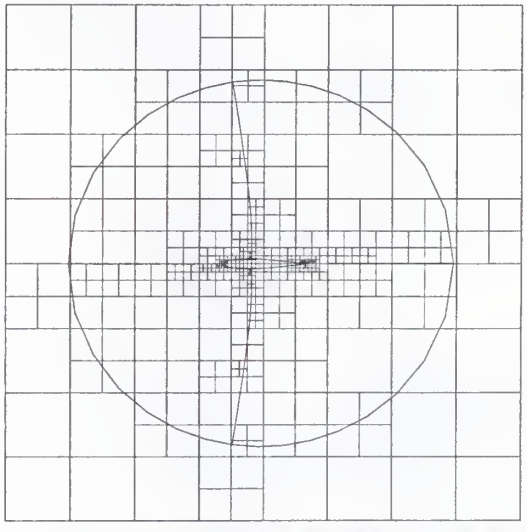


Figure 2.1: Example quad tree mesh

The PM tree is expensive to construct and would be very inefficient to use if

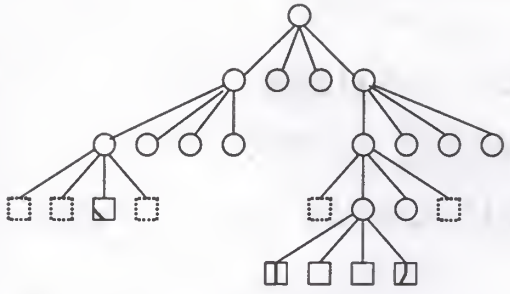


Figure 2.2: Example PM tree structure

it had to be reconstructed each time a grid moved. Therefore, for each dynamic group, a set of transformations are maintained between the current position and the original position in which the PM tree was built. Whenever the PM tree is used to find an interpolation stencil in one grid for a grid point in another grid, the transformations are used to transform the grid point to the original coordinate system. The transformed grid point can then be used with the PM tree constructed in the original coordinate system of the grids. Thus the PM tree must be constructed only once.

Interpolation Stencils

The primary function of the PM tree is to help find interpolation stencils for grid points which require interpolated flow field information. When an interpolation stencil is required, the PM tree is used to classify the corresponding grid point relative to each superblock. This quickly identifies which superblocks and grids the grid point lies in and therefore which superblocks might offer a source of interpolation information. This answers the in/out question directly. However, once a point is identified as being inside a given superblock, the exact curvilinear coordinates corresponding to the Cartesian coordinates of the grid point must be found.

For a curvilinear grid defined by the coordinates of the intersections of three families of boundary conforming grid lines denoted by (ξ, η, ζ), the coordinates at

any point within a cell can be calculated from tri-linear interpolation

$$\begin{aligned}\boldsymbol{R}(\xi,\eta,\zeta) = {} & (1-u)(1-v)(1-w)\,\boldsymbol{r}(I,J,K) + \\ & (1-u)(1-v)w\,\boldsymbol{r}(I,J,K+1) + \\ & (1-u)v(1-w)\,\boldsymbol{r}(I,J+1,K) + \\ & (1-u)vw\,\boldsymbol{r}(I,J+1,K+1) + \\ & u(1-v)(1-w)\,\boldsymbol{r}(I+1,J,K) + \\ & u(1-v)w\,\boldsymbol{r}(I+1,J,K+1) + \\ & uv(1-w)\,\boldsymbol{r}(I+1,J+1,K) + \\ & uvw\,\boldsymbol{r}(I+1,J+1,K+1) \end{aligned} \tag{2.1}$$

where $\boldsymbol{r}(I,J,K), \boldsymbol{r}(I+1,J,K), \ldots$ are the known coordinates of the eight corners of a cell. The index (I,J,K) denotes the three dimensional equivalent of the lower left corner of a two dimensional cell; while, (u,v,w) vary between 0 and 1 throughout the cell so that

$$\begin{aligned} \xi &= I+u, & I &= 1,2,\ldots,NI-1, & 0 \le u \le 1 \\ \eta &= J+v, & J &= 1,2,\ldots,NJ-1, & 0 \le v \le 1 \\ \zeta &= K+w, & K &= 1,2,\ldots,NK-1, & 0 \le w \le 1 \end{aligned} \tag{2.2}$$

and $\boldsymbol{R}(\xi,\eta,\zeta)$ is a piecewise continuous function over the entire grid.

For every known point $\boldsymbol{r}$ that lies within a grid, there exists some (ξ,η,ζ) such that $\boldsymbol{r} = \boldsymbol{R}(\xi,\eta,\zeta)$. However, in order to find (ξ,η,ζ) that corresponds to a known $\boldsymbol{r}$, the nonlinear function $\boldsymbol{F} = \boldsymbol{R}(\xi,\eta,\zeta) - \boldsymbol{r}$ must be minimized. Newton's method can be used to minimize this function iteratively using

$$\boldsymbol{\xi}^{m+1} = \boldsymbol{\xi}^m - \left[\frac{\partial \boldsymbol{F}^m}{\partial \boldsymbol{\xi}^m}\right]^{-1} \boldsymbol{F}^m \tag{2.3}$$

where $\boldsymbol{\xi}$ is the curvilinear coordinate vector (ξ,η,ζ), m is the Newton iteration

counter, and the jacobian matrix is calculated from the equations

$$\begin{aligned}
\frac{\partial \boldsymbol{F}}{\partial \xi} &= C_1 + vC_3 + w\left[C_5 + vC_7\right] \\
\frac{\partial \boldsymbol{F}}{\partial \eta} &= C_2 + uC_3 + w\left[C_6 + uC_7\right] \\
\frac{\partial \boldsymbol{F}}{\partial \zeta} &= C_4 + uC_5 + v\left[C_6 + uC_7\right]
\end{aligned} \tag{2.4}$$

where

$$\begin{aligned}
C_1 =& \boldsymbol{r}(I+1,J,K) - \boldsymbol{r}(I,J,K) \\
C_2 =& \boldsymbol{r}(I,J+1,K) - \boldsymbol{r}(I,J,K) \\
C_3 =& \boldsymbol{r}(I+1,J+1,K) - \boldsymbol{r}(I,J+1,K) - C_1 \\
C_4 =& \boldsymbol{r}(I,J,K+1) - \boldsymbol{r}(I,J,K) \\
C_5 =& \boldsymbol{r}(I+1,J,K+1) - \boldsymbol{r}(I,J,K+1) - C_1 \\
C_6 =& \boldsymbol{r}(I,J+1,K+1) - \boldsymbol{r}(I,J,K+1) - C_2 \\
C_7 =& \boldsymbol{r}(I+1,J+1,K+1) - \boldsymbol{r}(I,J+1,K+1) - \\
& \boldsymbol{r}(I+1,J,K+1) + \boldsymbol{r}(I,J,K+1) - C_3
\end{aligned} \tag{2.5}$$

Newton's method needs a good starting point; therefore, stored in the leaf nodes of the octree and the BSP trees are curvilinear coordinates at which to start the search. Although the PM tree classifies a point relative to a superblock, a starting point identifies a particular cell within a particular grid of the superblock. If the octree is sufficiently refined, the starting point should be close enough to ensure that stencil jumping will converge. As the curvilinear coordinates $\boldsymbol{\xi}$ are updated with equation 2.3, if $\Delta\boldsymbol{\xi}$ exceeds the range of $0 \rightarrow 1$ then the search proceeds to a neighboring cell and the jacobian matrix, as well as the corners of the containing cell, must be updated. This algorithm is commonly called *stencil jumping.*

Hole Cutting

Beggar uses an outline and fill algorithm for cutting holes. In this algorithm, the facets of the hole cutting surface are used to create an outline of the hole. The cells of a grid through which a hole cutting facet passes are located by using the PM tree to locate the cells containing the vertices of the facet. These cells are compared to the facet and are marked as being either on the *hole* side or the *world* side of the hole cutting surface. If the cells containing the facet vertices are not neighbors, the facet is subdivided recursively and new points on the hole cutting facet are introduced. These new points are processed just like the original facet vertices to ensure a continuous outline of the hole cutting surface. Once the complete hole cutting surface is outlined, the hole is flood filled by sweeping through the grid and marking as hole points any points that lie between hole points or between a grid boundary and a hole point. The marking of holes is capped off by the world side points created from the outline. This process is able to mark holes without comparing every grid point against each hole cutting surface and it places no special restrictions on how the hole cutting surfaces are defined as long as they are completely closed. It also allows holes to be cut using infinitely thin surfaces.

During the search for interpolation stencils, it is possible that a stencil may be found that is in someway undesirable. If no other interpolation stencil can be found for this point, then the point is marked out and an attempt is made to find an interpolation stencil for a neighboring point. This process essentially grows the hole in an attempt to find a valid grid assembly.

There are several weaknesses in this hole cutting algorithm. During the flood fill, if the hole outline is not completely surrounded by world side points, a *leaky* hole can result and the complete grid can be marked out. Conversely, the use of recursive subdivision of facets to ensure that a complete hole is outlined can dramatically increase execution time when hole cutting surfaces cut across a singularity or a region of viscous spacing. In such cases, it is possible to coarsely outline the hole and to

use the natural process of marking out points which fail interpolation rather than doing the flood fill. This option is often referred to as the "nofill" option based on the command line argument that is used to invoke this option and the fact that the holes are outlined but are not filled.

Donors and Receptors

One of the more important concepts is how to handle block-to-block and overlapped communications. Beggar introduces the concept of *donors* and *receptors* to deal with the communication introduced by these two boundary conditions. Since the flow solver uses a finite volume discretization, flow field information is associated with the grid cells or cell centers. A receptor will grab flow field information from one cell and store it in another cell. The receptor only needs to know which grid and cell from which to get the information. A donor will interpolate flow field information from a cell and then put the interpolated data into another storage location. The donor needs to know the grid from which to interpolate data, as well as an interpolation stencil for use in interpolating data from eight surrounding cell centers. Thus, block-to-block connections can be implemented using only receptors. Overlapped connections are implemented with donors.

If all of the grids' data are stored in core, a donor can be used to interpolate the flow data from one grid and to store the interpolated values into another grid. However, if all of the grids' data are not available, a small, donor value array (DVA) is needed to store the intermediate values. A donor, associated with the source grid, is used to perform the interpolation and to store the result into the DVA. Then a receptor, associated with the destination grid, is used to fetch the values from the DVA and store it into the final location.

Boundary Condition Identification

The automatic identification of most of the boundary conditions centers around several interdependent linked lists. The first of these is a list of the points on the boundaries of the grids in each superblock. A tolerance is used to decide if two points are coincident, so that the list contains only one entry for each unique boundary point. Another tolerance is used to decide if a point lies on a user specified reflection plane. Another list is constructed using the unique cell faces on each grid's boundaries. While building this list, degenerate cell faces and cell faces that lie on a reflection plane are identified. The order of the forming points for a cell face is not important for identification, therefore the forming points are sorted using pointers into the points list. The cell faces can then be associated with the first point in its sorted list of forming points. For a finite volume code, each cell face on a block-to-block boundary connects exactly two cells from either the same grid or from two different grids. Thus, for a given boundary point, if its list of associated cell faces contains two faces that are built from the same forming points, a block-to-block connection is defined.

CHAPTER 3
FLOW SOLUTION

Although the flow solver is not the focus of this work, this section is included for completeness. The flow solution algorithm supplies some opportunities for parallelization that affect the total performance of the code. The governing equations are presented, the unique solution algorithms are presented, and the general numerical solution techniques are presented.

Governing Equations

The equations governing the motion of a fluid are the statements of the conservation of mass, momentum, and energy. As an example, Newton's second law of motion describes the conservation of momentum. However, Newton's second law, as presented in most dynamics textbooks, is written to describe the motion of a particle, a rigid body, or a system of particles, that is, a well defined quantity of mass whose motion is described in a Lagrangian reference frame. For the motion of a fluid, it is often more useful to consider flow through a region of space or a control volume (an Eulerian reference frame). Considering a control volume $V(t)$ that is bounded by a surface $S(t)$, Reynolds' Transport Theorem (see Potter and Foss [43, pages 72–87] for an example of the derivation) is used to convert the time rate of change of an extensive property of a system into integrals over the control volume, i.e.

$$\frac{d}{dt}\int_{V(t)} \phi(\boldsymbol{x},t)dV = \int_{V(t)} \frac{\partial \phi}{\partial t}dV + \oint_{S(t)} \phi \boldsymbol{u}\cdot\hat{\boldsymbol{n}}dS \tag{3.1}$$

These two terms represent a variation in the conserved property within a volume $V(t)$ due to some internal sources (the volume integral) and a variation due to flux

across the boundary surface $S(t)$ (the surface integral). The variable ϕ represents any conserved quantity (such as $\rho, \rho\boldsymbol{u}, \rho E$ for mass, linear momentum, and total energy, all per unit volume), $\boldsymbol{u}$ is a local velocity vector, and $\hat{\boldsymbol{n}}$ is the unit vector normal to dS. The surface integral is converted to a volume integral using the vector form of Gauss's Theorem

$$\oint_S \phi\boldsymbol{u} \cdot \hat{\boldsymbol{n}} dS = \int_V \boldsymbol{\nabla} \cdot (\phi\boldsymbol{u})\, dV \tag{3.2}$$

which assumes that $\boldsymbol{\nabla} \cdot \boldsymbol{u}$ exists everywhere in V. Thus, the time rate of change of the conserved property can be written as

$$\frac{d}{dt} \int_{V(t)} \phi(\boldsymbol{x}, t) dV = \int_{V(t)} \frac{\partial \phi}{\partial t} + \boldsymbol{\nabla} \cdot (\phi\boldsymbol{u}) dV \tag{3.3}$$

The time rate of change of the conserved quantity is dependent upon source terms that can act on the volume or on the surface of the volume. If we can represent the source terms by a volume integral of a scalar quantity ψ and a surface integral of a vector quantity $\boldsymbol{\psi}$, the general conservation law can be written as

$$\int_{V(t)} \frac{\partial \phi}{\partial t} + \boldsymbol{\nabla} \cdot (\phi\boldsymbol{u}) dV = \int_{V(t)} \psi + \boldsymbol{\nabla} \cdot \boldsymbol{\psi} dV \tag{3.4}$$

Since an arbitrary volume is assumed, the integrand must apply for an infinitesimal volume. The integral can be removed to yield the differential form

$$\frac{\partial \phi}{\partial t} + \boldsymbol{\nabla} \cdot (\phi\boldsymbol{u}) = \psi + \boldsymbol{\nabla} \cdot \boldsymbol{\psi} \tag{3.5}$$

For the conservation of mass, mass is conserved and there are no source terms. Replacing ϕ by the density ρ in equation 3.5, the differential, continuity equation is

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot (\rho\boldsymbol{u}) = 0 \tag{3.6}$$

or, written in Cartesian coordinates

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u)}{\partial x} + \frac{\partial (\rho v)}{\partial y} + \frac{\partial (\rho w)}{\partial z} = 0 \tag{3.7}$$

where u, v, and w are the three Cartesian components of the velocity.

For the conservation of momentum, the source terms are the forces acting on the control volume. Ignoring the gravitational and inertial forces, the sum of the forces acting on the system can be written as

$$\sum \boldsymbol{F} = -\int_{S(t)} p\hat{\boldsymbol{n}} dS + \int_{S(t)} \boldsymbol{\tau} dS \tag{3.8}$$

where $\boldsymbol{\tau}$ is the viscous stress vector and p is the pressure. Note that the pressure has been separated from the viscous stress, whereas it is often included as a spherical stress term. The differential form is

$$\begin{aligned} F_x &= -\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} \\ F_y &= -\frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} \\ F_z &= -\frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} \end{aligned} \tag{3.9}$$

where τ_{xx}, τ_{xy}, etc. are elements of the viscous stress tensor (see Potter and Foss [43, pages 171–174] for the derivation). This tensor is symmetric so that $\tau_{yx} = \tau_{xy}$, $\tau_{zx} = \tau_{xz}$, and $\tau_{zy} = \tau_{yz}$. Using these equations as the source terms and substituting $\rho\boldsymbol{u}$ into equation 3.5 as the conserved quantity, the three Cartesian components of the conservation of momentum are

$$\begin{aligned} \frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho u^2 + p)}{\partial x} + \frac{\partial(\rho uv)}{\partial y} + \frac{\partial(\rho uw)}{\partial z} &= \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial \tau_{xz}}{\partial z} \\ \frac{\partial(\rho v)}{\partial t} + \frac{\partial(\rho uv)}{\partial x} + \frac{\partial(\rho v^2 + p)}{\partial y} + \frac{\partial(\rho vw)}{\partial z} &= \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{yz}}{\partial z} \\ \frac{\partial(\rho w)}{\partial t} + \frac{\partial(\rho uw)}{\partial x} + \frac{\partial(\rho vw)}{\partial y} + \frac{\partial(\rho w^2 + p)}{\partial z} &= \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} \end{aligned} \tag{3.10}$$

where the pressure terms have been moved to the left hand side. Formally, these equations are the Navier-Stokes equations. However, in general, the term Navier-Stokes equations is used to refer to the complete set of conservation laws when the viscous terms are included as shown here.

For the conservation of energy, the source terms are the rate of heat transfer to the system minus the rate of work done by the system. Substituting ρE into equation

3.4, the conservation of energy is written as

$$\int_{V(t)} \frac{\partial(\rho E)}{\partial t} + \nabla \cdot (\rho E \boldsymbol{u}) dV = \dot{Q} - \dot{W} \tag{3.11}$$

or, in differential form

$$\frac{\partial(\rho E)}{\partial t} + \nabla \cdot (\rho E \boldsymbol{u}) = \dot{Q} - \dot{W} \tag{3.12}$$

Ignoring any internal heat sources, the heat transfer rate can be written as

$$\dot{Q} = -\int_{S(t)} \bar{\boldsymbol{q}} \cdot \hat{\boldsymbol{n}} dS \tag{3.13}$$

where $\bar{\boldsymbol{q}}$ is the heat flux vector. This integral can be converted to a volume integral and then written in the differential form

$$\dot{Q} = -\frac{\partial \bar{q}_x}{\partial x} - \frac{\partial \bar{q}_y}{\partial y} - \frac{\partial \bar{q}_z}{\partial z} \tag{3.14}$$

The work rate is due to the forces acting on the surface of the control volume. Ignoring any work by gravitational or inertia forces, the work rate is written in the form

$$\dot{W} = \int_{S(t)} p\boldsymbol{u} \cdot \hat{\boldsymbol{n}} dS - \int_{S(t)} \boldsymbol{\tau} \cdot \boldsymbol{u} dS \tag{3.15}$$

The differential form is

$$\begin{aligned} \dot{W} = &\frac{\partial pu}{\partial x} + \frac{\partial pv}{\partial y} + \frac{\partial pw}{\partial z} - \tau_{xx}\frac{\partial u}{\partial x} - \tau_{yy}\frac{\partial v}{\partial y} - \tau_{zz}\frac{\partial w}{\partial z} - \\ &\tau_{xy}\left[\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right] - \tau_{xz}\left[\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right] - \tau_{yz}\left[\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right] \end{aligned} \tag{3.16}$$

Plugging equations 3.14 and 3.16 into equation 3.12 yields the final differential form of the conservation of energy equation

$$\begin{aligned} &\frac{\partial(\rho E)}{\partial t} + \frac{\partial u(\rho E + p)}{\partial x} + \frac{\partial v(\rho E + p)}{\partial y} + \frac{\partial w(\rho E + p)}{\partial w} = -\frac{\partial \bar{q}_x}{\partial x} - \frac{\partial \bar{q}_y}{\partial y} - \frac{\partial \bar{q}_z}{\partial z} + \\ &\frac{\partial\left(\tau_{xx}u + \tau_{xy}v + \tau_{xz}w\right)}{\partial x} + \frac{\partial\left(\tau_{xy}u + \tau_{yy}v + \tau_{yz}w\right)}{\partial y} + \frac{\partial\left(\tau_{xz}u + \tau_{yz}v + \tau_{zz}w\right)}{\partial z} \end{aligned} \tag{3.17}$$

Counting the primitive variables ρ, u, v, w, E, p, and T and the 6 unique elements of the viscous stress tensor, there are 13 unknowns and only 5 equations (the conservation laws). In order to make a solution tractable, 8 more equations are needed.

Fortunately, a relationship between the components of the stress tensor and the velocity gradients is possible. The velocity gradients are directly related to the rate-of-strain tensor and the vorticity tensor. Constitutive equations then define the relationship between the stress components and the rate-of-strain components. For a Newtonian fluid, a linear relationship between the stress and the strain rate is assumed. Since the strain rate tensor is symmetric, there are only 6 unique strain rate components. Assuming a linear relationship between the 6 unique stress components and the 6 unique strain rate components, there are 36 material constants that must be defined. The assumption of an isotropic material reduces this to 2 constants. For fluids, these two constants are the dynamic viscosity μ and the second coefficient of viscosity λ. From Stoke's hypothesis, the relationship

$$\lambda = -\frac{2}{3}\mu \tag{3.18}$$

can be used for the compressible flow of air. For a Newtonian, isotropic fluid, the final relationships between the components on the stess tensor and the velocity gradients are

$$\begin{aligned}
\tau_{xx} &= \frac{2}{3}\mu\left(2\frac{\partial u}{\partial x} - \frac{\partial v}{\partial y} - \frac{\partial w}{\partial z}\right) \\
\tau_{yy} &= \frac{2}{3}\mu\left(-\frac{\partial u}{\partial x} + 2\frac{\partial v}{\partial y} - \frac{\partial w}{\partial z}\right) \\
\tau_{zz} &= \frac{2}{3}\mu\left(-\frac{\partial u}{\partial x} - \frac{\partial v}{\partial y} + 2\frac{\partial w}{\partial z}\right) \\
\tau_{xy} &= \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \\
\tau_{xz} &= \mu\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \\
\tau_{yz} &= \mu\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right)
\end{aligned} \tag{3.19}$$

With the original 5 conservation equations and the 6 relationships between the viscous stesses and the velocity gradients, only 2 more equations are needed. If a perfect gas is assumed, the thermodynamic state can be specified by only two thermodynamic variables. If p and ρ are choosen as the two independent thermodynamic variables, the perfect gas law

$$p = \rho \mathrm{R} T \tag{3.20}$$

(where R is the gas constant) can be used to calculate the temperature T. The relationship for the internal energy e per unit mass is

$$e = \frac{p}{\rho(\gamma - 1)} \tag{3.21}$$

where γ is the ratio of specific heats ($\gamma = 1.4$ for air) and the total energy per unit mass is related to the internal energy by

$$E = e + \frac{1}{2} U^2 \tag{3.22}$$

where U is the magnitude of the velocity vector $\boldsymbol{u}$.

Vector Form

The three conservation laws, written in differential form, can be combined in the vector differential equation

$$\frac{\partial \boldsymbol{q}}{\partial t} + \frac{\partial \boldsymbol{f}_i}{\partial x} + \frac{\partial \boldsymbol{g}_i}{\partial y} + \frac{\partial \boldsymbol{h}_i}{\partial z} = \frac{\partial \boldsymbol{f}_v}{\partial x} + \frac{\partial \boldsymbol{g}_v}{\partial y} + \frac{\partial \boldsymbol{h}_v}{\partial z} \tag{3.23}$$

where

$$\boldsymbol{q} = \begin{Bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{Bmatrix}, \quad \boldsymbol{f}_i = \begin{Bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ u(\rho E + p) \end{Bmatrix}, \quad \boldsymbol{f}_v = \begin{Bmatrix} 0 \\ \tau_{xx} \\ \tau_{xy} \\ \tau_{xz} \\ u\tau_{xx} + v\tau_{xy} + w\tau_{xz} - \bar{q}_x \end{Bmatrix}$$

$$\boldsymbol{g}_i = \begin{Bmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ \rho vw \\ v(\rho E + p) \end{Bmatrix}, \quad \boldsymbol{g}_v = \begin{Bmatrix} 0 \\ \tau_{xy} \\ \tau_{yy} \\ \tau_{yz} \\ u\tau_{xy} + v\tau_{yy} + w\tau_{yz} - \bar{q}_y \end{Bmatrix}$$

$$\boldsymbol{h}_i = \begin{Bmatrix} \rho w \\ \rho wu \\ \rho wv \\ \rho w^2 + p \\ w(\rho E + p) \end{Bmatrix}, \quad \boldsymbol{h}_v = \begin{Bmatrix} 0 \\ \tau_{xz} \\ \tau_{yz} \\ \tau_{zz} \\ u\tau_{xz} + v\tau_{yz} + w\tau_{zz} - \bar{q}_z \end{Bmatrix} \tag{3.24}$$

The first component of the vector equation represents the conservation of mass. The next three components represent the conservation of momentum. The fifth component represents the conservation of energy.

The terms f_i, g_i, and h_i represent the inviscid flux vectors and f_v, g_v, and h_v represent the viscous flux vectors. Setting $f_v = g_v = h_v = 0$ recovers the Euler equations, which govern inviscid fluid flow. The elements of the vector $\boldsymbol{q}$ are the conserved variables, as opposed to the primitive variables ρ, u, v, w, and p.

The use of subscripts on the terms $\bar{q}_x$, $\bar{q}_y$, and $\bar{q}_z$ represents the components of the heat transfer vector as opposed to partial derivative notation. Considering only heat conduction, Fourier's law can be used to relate the heat flux vector to the

temperature gradient

$$\bar{q} = -k\nabla T \tag{3.25}$$

where k is the heat conductivity and T is the temperature. The Prandtl number, defined as

$$Pr = \frac{c_p \mu}{k} \tag{3.26}$$

is used to compute the heat conductivity k from the viscosity μ (for air at standard conditions, $Pr = 0.72$). Using the relationship

$$c_p = \frac{\gamma \mathrm{R}}{\gamma - 1} \tag{3.27}$$

for a perfect gas, the components of the heat flux vector can be written as

$$\begin{aligned} \bar{q}_x &= -\frac{\gamma \mathrm{R}}{\gamma - 1}\frac{\mu}{Pr}\frac{\partial T}{\partial x} \\ \bar{q}_y &= -\frac{\gamma \mathrm{R}}{\gamma - 1}\frac{\mu}{Pr}\frac{\partial T}{\partial y} \\ \bar{q}_z &= -\frac{\gamma \mathrm{R}}{\gamma - 1}\frac{\mu}{Pr}\frac{\partial T}{\partial z} \end{aligned} \tag{3.28}$$

Non-Dimensionalization

The governing equations are non-dimensionalized by freestream conditions so that

$$\begin{gathered} \tilde{\rho} = \frac{\rho}{\rho_\infty}, \quad \tilde{u} = \frac{u}{a_\infty}, \quad \tilde{v} = \frac{v}{a_\infty}, \quad \tilde{w} = \frac{w}{a_\infty}, \quad \tilde{p} = \frac{p}{\rho_\infty a_\infty^2}, \quad \tilde{E} = \frac{E}{a_\infty^2} \\ \tilde{x} = \frac{x}{L}, \quad \tilde{y} = \frac{y}{L}, \quad \tilde{z} = \frac{z}{L}, \quad \tilde{\mu} = \frac{\mu}{\mu_\infty}, \quad \tilde{t} = \frac{t a_\infty}{L} \end{gathered} \tag{3.29}$$

where the $\tilde{}$ denotes non-dimensional quantities, the subscript ∞ denotes freestream conditions, L is some dimensional reference length and a is the speed of sound, which is defined by the relations

$$a = \sqrt{\gamma \frac{p}{\rho}} = \sqrt{\gamma \mathrm{R} T} \tag{3.30}$$

The Mach number is the non-dimensional velocity. The freestream Mach number is

$$\mathrm{M}_\infty = \frac{U_\infty}{a_\infty} \tag{3.31}$$

where U_∞ is the magnitude of the freestream velocity. Therefore, the non-dimensional velocity components become

$$\tilde{u} = \frac{u}{U_\infty}\mathrm{M}_\infty, \quad \tilde{v} = \frac{v}{U_\infty}\mathrm{M}_\infty, \quad \tilde{w} = \frac{w}{U_\infty}\mathrm{M}_\infty \tag{3.32}$$

The terms u/U_∞, etc. represent scaled direction cosines; therefore, the non-dimensional velocities are scaled values of the Mach number.

The Reynolds number is the non-dimensional parameter

$$Re = \frac{\rho_\infty U_\infty L}{\mu_\infty} \tag{3.33}$$

which arises from the non-dimensionalization of the conservation of momentum equation. This parameter represents the ratio of inertia forces to viscous forces.

The non-dimensional governing equations can be written in the same form as equations 3.23 and 3.24 by replacing the dimensional quantities by the corresponding non-dimensional quantities. However, in the process of non-dimensionalizing the equations, the non-dimensional term M_∞/Re arises from the viscous flux vectors. Therefore, the definition of the viscous stresses and the heat flux components must be modified as

$$\begin{aligned}
\tau_{xx} &= \frac{2}{3}\tilde{\mu}\frac{\mathrm{M}_\infty}{Re}\left(2\frac{\partial\tilde{u}}{\partial\tilde{x}} - \frac{\partial\tilde{v}}{\partial\tilde{y}} - \frac{\partial\tilde{w}}{\partial\tilde{z}}\right) \\
\tau_{yy} &= \frac{2}{3}\tilde{\mu}\frac{\mathrm{M}_\infty}{Re}\left(-\frac{\partial\tilde{u}}{\partial\tilde{x}} + 2\frac{\partial\tilde{v}}{\partial\tilde{y}} - \frac{\partial\tilde{w}}{\partial\tilde{z}}\right) \\
\tau_{zz} &= \frac{2}{3}\tilde{\mu}\frac{\mathrm{M}_\infty}{Re}\left(-\frac{\partial\tilde{u}}{\partial\tilde{x}} - \frac{\partial\tilde{v}}{\partial\tilde{y}} + 2\frac{\partial\tilde{w}}{\partial\tilde{z}}\right) \\
\tau_{xy} &= \tilde{\mu}\frac{\mathrm{M}_\infty}{Re}\left(\frac{\partial\tilde{u}}{\partial\tilde{y}} + \frac{\partial\tilde{v}}{\partial\tilde{x}}\right) \\
\tau_{xz} &= \tilde{\mu}\frac{\mathrm{M}_\infty}{Re}\left(\frac{\partial\tilde{u}}{\partial\tilde{z}} + \frac{\partial\tilde{w}}{\partial\tilde{x}}\right) \\
\tau_{yz} &= \tilde{\mu}\frac{\mathrm{M}_\infty}{Re}\left(\frac{\partial\tilde{v}}{\partial\tilde{z}} + \frac{\partial\tilde{w}}{\partial\tilde{y}}\right)
\end{aligned} \tag{3.34}$$

and

$$\tilde{q}_x = -\frac{1}{\gamma - 1}\frac{\tilde{\mu}}{Pr}\frac{\mathrm{M}_\infty}{Re}\frac{\partial \tilde{T}}{\partial \tilde{x}}$$
$$\tilde{q}_y = -\frac{1}{\gamma - 1}\frac{\tilde{\mu}}{Pr}\frac{\mathrm{M}_\infty}{Re}\frac{\partial \tilde{T}}{\partial \tilde{y}}$$
$$\tilde{q}_z = -\frac{1}{\gamma - 1}\frac{\tilde{\mu}}{Pr}\frac{\mathrm{M}_\infty}{Re}\frac{\partial \tilde{T}}{\partial \tilde{z}} \tag{3.35}$$

The non-dimensional equation of state becomes

$$\tilde{p} = \frac{\tilde{\rho}\tilde{T}}{\gamma} \tag{3.36}$$

and the non-dimensional energy is related to the non-dimensional density and pressure by the equation

$$\tilde{E} = \frac{\tilde{p}}{\tilde{\rho}(\gamma - 1)} + \frac{1}{2}(\tilde{u}^2 + \tilde{v}^2 + \tilde{w}^2) \tag{3.37}$$

The non-dimensional viscosity coefficient is related to the non-dimensional temperature by the power law

$$\tilde{\mu} = \tilde{T}^{2/3} \tag{3.38}$$

Coordinate Transformation

The use of a general, boundary conforming, structured grid introduces an ordered set of grid points represented by Cartesian coordinates given at the integer curvilinear coordinates ξ, η, ζ. In order to solve the governing equations in this curvilinear coordinate system, the partial derivatives with respect to the Cartesian coordinates must be converted into partial derivatives with respect to the curvilinear coordinates. This requires a transformation of the form

$$\xi = \xi(x, y, z, t), \quad \eta = \eta(x, y, z, t), \quad \zeta = \zeta(x, y, z, t), \quad \tau = \tau(t) \tag{3.39}$$

Applying the chain rule, the partial derivatives with respect to the Cartesian coordinates can be written as

$$\begin{aligned}
\frac{\partial}{\partial x} &= \xi_x \frac{\partial}{\partial \xi} + \eta_x \frac{\partial}{\partial \eta} + \zeta_x \frac{\partial}{\partial \zeta} \\
\frac{\partial}{\partial y} &= \xi_y \frac{\partial}{\partial \xi} + \eta_y \frac{\partial}{\partial \eta} + \zeta_y \frac{\partial}{\partial \zeta} \\
\frac{\partial}{\partial z} &= \xi_z \frac{\partial}{\partial \xi} + \eta_z \frac{\partial}{\partial \eta} + \zeta_z \frac{\partial}{\partial \zeta} \\
\frac{\partial}{\partial t} &= \tau_t \frac{\partial}{\partial \tau} + \xi_t \frac{\partial}{\partial \xi} + \eta_t \frac{\partial}{\partial \eta} + \zeta_t \frac{\partial}{\partial \zeta}
\end{aligned} \tag{3.40}$$

where the term ξ_x represents the partial derivative of ξ with respect to x, etc. Thus, the metric term ξ_x represents the variation in ξ with a unit variation in x while y, z, and t are held constant. These terms are not easily evaluated. However, the partial derivatives of the Cartesian coordinates with respect to the curvilinear coordinates that arise from the inverse transformation represented by

$$x = x(\xi, \eta, \zeta, \tau), \quad y = y(\xi, \eta, \zeta, \tau), \quad z = z(\xi, \eta, \zeta, \tau), \quad t = t(\tau) \tag{3.41}$$

are easily evaluated. Applying the chain rule again, the partial derivatives with respect to the curvilinear coordinates can be written as

$$\begin{aligned}
\frac{\partial}{\partial \xi} &= x_\xi \frac{\partial}{\partial x} + y_\xi \frac{\partial}{\partial y} + z_\xi \frac{\partial}{\partial z} \\
\frac{\partial}{\partial \eta} &= x_\eta \frac{\partial}{\partial x} + y_\eta \frac{\partial}{\partial y} + z_\eta \frac{\partial}{\partial z} \\
\frac{\partial}{\partial \zeta} &= x_\zeta \frac{\partial}{\partial x} + y_\zeta \frac{\partial}{\partial y} + z_\zeta \frac{\partial}{\partial z} \\
\frac{\partial}{\partial \tau} &= t_\tau \frac{\partial}{\partial t} + x_\tau \frac{\partial}{\partial x} + y_\tau \frac{\partial}{\partial y} + z_\tau \frac{\partial}{\partial z}
\end{aligned} \tag{3.42}$$

Comparing equations 3.40 and 3.42 the Jacobian matrix of the transformation 3.39 is seen to be the inverse of the Jacobian of the inverse transformation 3.41 (see Belk

[44, appendix A] for a complete derivation). This yields the relationships

$$\begin{aligned}
\xi_x &= (y_\eta z_\zeta - z_\eta y_\zeta)/J \\
\xi_y &= (z_\eta x_\zeta - x_\eta z_\zeta)/J \\
\xi_z &= (x_\eta y_\zeta - y_\eta x_\zeta)/J \\
\xi_t &= -\tau_t x_\tau \xi_x - \tau_t y_\tau \xi_y - \tau_t z_\tau \xi_z \\
\eta_x &= (z_\xi y_\zeta - y_\xi z_\zeta)/J \\
\eta_y &= (x_\xi z_\zeta - z_\xi x_\zeta)/J \\
\eta_z &= (y_\xi x_\zeta - x_\xi y_\zeta)/J \\
\eta_t &= -\tau_t x_\tau \eta_x - \tau_t y_\tau \eta_y - \tau_t z_\tau \eta_z \\
\zeta_x &= (y_\xi z_\eta - z_\xi y_\eta)/J \\
\zeta_y &= (z_\xi x_\eta - x_\xi z_\eta)/J \\
\zeta_z &= (x_\xi y_\eta - y_\xi x_\eta)/J \\
\zeta_t &= -\tau_t x_\tau \zeta_x - \tau_t y_\tau \zeta_y - \tau_t z_\tau \zeta_z
\end{aligned} \tag{3.43}$$

where J is the determinant of the Jacobian matrix of the inverse transformation

$$J = x_\xi(y_\eta z_\zeta - z_\eta y_\zeta) - y_\xi(x_\eta z_\zeta - z_\eta x_\zeta) + z_\xi(x_\eta y_\zeta - y_\eta x_\zeta) \tag{3.44}$$

The governing equations are then written in the form

$$\frac{\partial \boldsymbol{Q}}{\partial \tau} + \frac{\partial \boldsymbol{F}_i - \boldsymbol{F}_v}{\partial \xi} + \frac{\partial \boldsymbol{G}_i - \boldsymbol{G}_v}{\partial \eta} + \frac{\partial \boldsymbol{H}_i - \boldsymbol{H}_v}{\partial \zeta} = 0 \tag{3.45}$$

where

$$\boldsymbol{Q} = J\boldsymbol{q}$$

$$\boldsymbol{F}_i = J(\boldsymbol{q}\xi_t + \boldsymbol{f}_i\xi_x + \boldsymbol{g}_i\xi_y + \boldsymbol{h}_i\xi_z)$$

$$\boldsymbol{G}_i = J(\boldsymbol{q}\eta_t + \boldsymbol{f}_i\eta_x + \boldsymbol{g}_i\eta_y + \boldsymbol{h}_i\eta_z)$$

$$\boldsymbol{H}_i = J(\boldsymbol{q}\zeta_t + \boldsymbol{f}_i\zeta_x + \boldsymbol{g}_i\zeta_y + \boldsymbol{h}_i\zeta_z)$$

$$\boldsymbol{F}_v = J(\boldsymbol{f}_v\xi_x + \boldsymbol{g}_v\xi_y + \boldsymbol{h}_v\xi_z)$$

$$\boldsymbol{G}_v = J(\boldsymbol{f}_v\eta_x + \boldsymbol{g}_v\eta_y + \boldsymbol{h}_v\eta_z)$$

$$\boldsymbol{H}_v = J(\boldsymbol{f}_v\zeta_x + \boldsymbol{g}_v\zeta_y + \boldsymbol{h}_v\zeta_z) \tag{3.46}$$

Flux Vector Splitting

The model hyperbolic equation is the one-dimensional linear convection equation

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \tag{3.47}$$

If $a > 0$, this equation describes the propogation of a wave in the $+x$ direction at the velocity a. The use of a backward time difference and a forward space difference or a central space difference to produce the explicit discretized finite difference equations

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + a\frac{u_{i+1}^n - u_i^n}{\Delta x} = 0 \tag{3.48}$$

and

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + a\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} = 0 \tag{3.49}$$

yields unconditionally unstable solution schemes. Instead, with $a > 0$, a backward space difference of the form

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + a\frac{u_i^n - u_{i-1}^n}{\Delta x} = 0 \tag{3.50}$$

is required to produce a stable scheme. Since the wave is propogating in the $+x$ direction, the backward space differencing represents "upwind" differencing. If the

wave speed a were negative, a forward space difference, again representing upwind differencing, would be required to produce a stable scheme.

The goal of "flux vector splitting" [45] is to split the flux vector into components which are associated with the positive and negative direction propogation of information so that upwind differencing can be used. This produces a stable scheme without the addition of any artificial dissipation that is often done with central difference schemes.

Consider the one-dimensional Euler equations in Cartesian coordinates

$$\frac{\partial \boldsymbol{Q}}{\partial t} + \frac{\partial \boldsymbol{F}}{\partial x} = 0 \tag{3.51}$$

Since the flux vector is a homogeneous function of degree one of Q, the governing equations can be written in quasi-linear form

$$\frac{\partial \boldsymbol{Q}}{\partial t} + A\frac{\partial \boldsymbol{Q}}{\partial x} = 0 \tag{3.52}$$

(this looks alot like the model equation). The matrix $A = \partial \boldsymbol{F}/\partial \boldsymbol{Q}$, is the flux Jacobian matrix. This matrix can be decomposed in the form

$$A = R\Lambda R^{-1} \tag{3.53}$$

where the columns of R are the right eigenvectors of A, the rows of R^{-1} are the left eigenvectors of A, and the matrix Λ is a diagonal matrix with the eigenvalues of A along the diagonal. The eigenvalues are of the form

$$\begin{aligned} \lambda_1 = \lambda_2 = \lambda_3 &= u \\ \lambda_4 &= u + a \\ \lambda_5 &= u - a \end{aligned} \tag{3.54}$$

where a is the speed of sound. For locally subsonic flow, some of the eigenvalues will be positive and some will be negative. Thus the matrix Λ can be written as

$$\Lambda = \Lambda^+ + \Lambda^- \tag{3.55}$$

where Λ^+ contains only the positive eigenvalues and Λ^- contains only the negative eigenvalues. Substituting this into equation 3.53, the Jacobian matrix is split into

$$\begin{aligned} A &= R\Lambda^+ R^{-1} + R\Lambda^- R^{-1} \\ &= A^+ + A^- \end{aligned} \tag{3.56}$$

and the split flux vectors are defined from the homogeneity property as

$$\begin{aligned} \boldsymbol{F}^+ &= A^+ \boldsymbol{Q} \\ \boldsymbol{F}^- &= A^- \boldsymbol{Q} \end{aligned} \tag{3.57}$$

so that

$$\boldsymbol{F} = \boldsymbol{F}^+ + \boldsymbol{F}^- \tag{3.58}$$

Upwind differencing is then used appropriately with the split flux vectors in the discretized governing equations. The complete form of the split flux vectors can be found in Hirsch [46].

An implicit discretization of the governing equations can be written as

$$\Delta \boldsymbol{Q}^{n+1} + \Delta\tau \left(\delta_\xi \boldsymbol{F}^{n+1} + \delta_\eta \boldsymbol{G}^{n+1} + \delta_\zeta \boldsymbol{H}^{n+1} \right) = 0 \tag{3.59}$$

where the superscript n denotes the time step,

$$\Delta \boldsymbol{Q}^{n+1} = \boldsymbol{Q}^{n+1}_{i,j,k} - \boldsymbol{Q}^{n}_{i,j,k} \tag{3.60}$$

and

$$\begin{aligned} \delta_\xi \boldsymbol{F} &= \frac{\boldsymbol{F}_{i+1/2,j,k} - \boldsymbol{F}_{i-1/2,j,k}}{\Delta\xi} \\ \delta_\eta \boldsymbol{G} &= \frac{\boldsymbol{G}_{i,j+1/2,k} - \boldsymbol{G}_{i,j-1/2,k}}{\Delta\eta} \\ \delta_\zeta \boldsymbol{H} &= \frac{\boldsymbol{H}_{i,j,k+1/2} - \boldsymbol{H}_{i,j,k-1/2}}{\Delta\zeta} \end{aligned} \tag{3.61}$$

For a finite volume discretization, the indices i, j, k represent a cell in which the dependent variables $\boldsymbol{Q}$ are assumed to be constant, and indices $i + 1/2, j, k$ and

$i-1/2, j, k$, for example, are opposing cell faces at which the flux is evaluated. The changes in the curvilinear coordinates $\Delta\xi$, $\Delta\eta$, and $\Delta\zeta$ are unity.

A first order time linearization of the flux terms [47, 48], leads to the equation

$$\frac{\Delta \boldsymbol{Q}^{n+1}}{\Delta\tau} + \delta_\xi \left[\boldsymbol{F}^n + \left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{Q}} \right)^n \Delta \boldsymbol{Q}^{n+1} \right] + \delta_\eta \left[\boldsymbol{G}^n + \left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{Q}} \right)^n \Delta \boldsymbol{Q}^{n+1} \right] + \delta_\zeta \left[\boldsymbol{H}^n + \left(\frac{\partial \boldsymbol{H}}{\partial \boldsymbol{Q}} \right)^n \Delta \boldsymbol{Q}^{n+1} \right] = 0 \tag{3.62}$$

Introducing the split flux vectors, produces the form

$$\begin{aligned} &\frac{\Delta \boldsymbol{Q}^{n+1}}{\Delta\tau} + \delta_\xi \left[\left(\frac{\partial \boldsymbol{F}^+}{\partial \boldsymbol{Q}} \right)^n \Delta \boldsymbol{Q}^{n+1} \right] + \delta_\xi \left[\left(\frac{\partial \boldsymbol{F}^-}{\partial \boldsymbol{Q}} \right)^n \Delta \boldsymbol{Q}^{n+1} \right] + \delta_\eta \left[\left(\frac{\partial \boldsymbol{G}^+}{\partial \boldsymbol{Q}} \right)^n \Delta \boldsymbol{Q}^{n+1} \right] \\ &+ \delta_\eta \left[\left(\frac{\partial \boldsymbol{G}^-}{\partial \boldsymbol{Q}} \right)^n \Delta \boldsymbol{Q}^{n+1} \right] + \delta_\zeta \left[\left(\frac{\partial \boldsymbol{H}^+}{\partial \boldsymbol{Q}} \right)^n \Delta \boldsymbol{Q}^{n+1} \right] + \delta_\zeta \left[\left(\frac{\partial \boldsymbol{H}^-}{\partial \boldsymbol{Q}} \right)^n \Delta \boldsymbol{Q}^{n+1} \right] \\ &= -\delta_\xi \left(\boldsymbol{F}^+ + \boldsymbol{F}^- \right)^n + \delta_\eta \left(\boldsymbol{G}^+ + \boldsymbol{G}^- \right)^n + \delta_\zeta \left(\boldsymbol{H}^+ + \boldsymbol{H}^- \right)^n \end{aligned} \tag{3.63}$$

or

$$\begin{aligned} &\frac{\Delta \boldsymbol{Q}^{n+1}}{\Delta\tau} + \delta_\xi \left[\left(A^+ \right)^n \left(\Delta \boldsymbol{Q}^+ \right)^{n+1} + \left(A^- \right)^n \left(\Delta \boldsymbol{Q}^- \right)^{n+1} \right] \\ &+ \delta_\eta \left[\left(B^+ \right)^n \left(\Delta \boldsymbol{Q}^+ \right)^{n+1} + \left(B^- \right)^n \left(\Delta \boldsymbol{Q}^- \right)^{n+1} \right] \\ &+ \delta_\eta \left[\left(C^+ \right)^n \left(\Delta \boldsymbol{Q}^+ \right)^{n+1} + \left(C^- \right)^n \left(\Delta \boldsymbol{Q}^- \right)^{n+1} \right] = -\boldsymbol{R}^n \end{aligned} \tag{3.64}$$

where

$$\boldsymbol{R}^n = \delta_\xi \left[\boldsymbol{F}^+ + \boldsymbol{F}^- \right]^n + \delta_\eta \left[\boldsymbol{G}^+ + \boldsymbol{G}^- \right]^n + \delta_\zeta \left[\boldsymbol{H}^+ + \boldsymbol{H}^- \right]^n \tag{3.65}$$

It should be noted that the Jacobian matrices A^+, A^-, etc. are not the same as the split flux Jacobian matrices $\boldsymbol{A}^+$, $\boldsymbol{A}^-$, etc. that were presented in equation 3.56. Instead, the notation A^+ is used to represent $\partial \boldsymbol{F}^+ / \partial \boldsymbol{Q}$, for example. This is required to preserve the conservation form of the equations. The final form of these Jacobian matrices, and the derivation thereof, can be found in Belk [44, appendix B].

In evaluating the split flux vectors at the cell faces according to the difference operators defined in equation 3.61, dependent variables from cells upwind of the cell face are used. For a first order spatial discretization, only the neighboring cell is used.

For second order accuracy, the dependent variables from the two neighboring cells are extrapolated to the cell face. As an example, the (+) flux is evaluated using cells to the left of the cell face

$$\boldsymbol{F}^{+}_{i+1/2,j,k} = \boldsymbol{F}^{+}\left(\boldsymbol{Q}^{L}_{i+1/2,j,k}\right) \tag{3.66}$$

where

$$\boldsymbol{Q}^{L}_{i+1/2,j,k} = \boldsymbol{Q}_{i,j,k} \tag{3.67}$$

for a first order accurate scheme, and

$$\boldsymbol{Q}^{L}_{i+1/2,j,k} = \frac{3}{2}\boldsymbol{Q}_{i,j,k} - \frac{1}{2}\boldsymbol{Q}_{i-1,j,k} \tag{3.68}$$

for a second order accurate scheme. Likewise, the (−) flux is evaluated using cells to the right of the cell face

$$\boldsymbol{F}^{-}_{i+1/2,j,k} = \boldsymbol{F}^{-}\left(\boldsymbol{Q}^{R}_{i+1/2,j,k}\right) \tag{3.69}$$

where

$$\boldsymbol{Q}^{R}_{i+1/2,j,k} = \boldsymbol{Q}_{i+1,j,k} \tag{3.70}$$

for a first order accurate scheme, and

$$\boldsymbol{Q}^{R}_{i+1/2,j,k} = \frac{3}{2}\boldsymbol{Q}_{i+1,j,k} - \frac{1}{2}\boldsymbol{Q}_{i+2,j,k} \tag{3.71}$$

for a second order accurate scheme. The extrapolation of the conserved variables to the cell face and their use to calculate the flux is often referred to as MUSCL extrapolation [49]. Alternatively, the primative variables can be extrapolated and used to calculate the flux or the flux can be evaluated at the cell centers and extrapolated to the cell center.

In the higher order schemes, flux limiters, applied to the flux, conserved variables, or the primitive variables, are used to selectively reduce the scheme to first order to avoid oscillations in the solution near discontinuities. The flux limiters available include the minmod, van Leer, and van Albada limiters.

Flux Difference Splitting

Hirsch [46] describes upwind methods as methods in which physical properties of the flow equations are introduced into the discretized equations. Flux vector splitting introduces the direction of propogation of information through consideration of the sign of the eigenvalues in the discretization. Another method that handles discontinuities well is due to Godunov [50]. In Godunov's method, the conserved variables are considered constant throughout each cell and a one-dimensional exact solution of the Euler equations is computed at each cell boundary. The two constant states on either side of a cell boundary define a Riemann (or shock tube) problem that can be solved exactly. An integral average of the exact solutions to the Riemann problems at each cell is taken to determine the solution at the next time step. Other methods have replaced the computationally expensive exact solution of the Riemann problem with an approximate Riemann solution. These methods, including the popular method due to Roe [7], are often referred to as "flux difference splitting" methods.

Considering the quasi-linear form of the one-dimensional Euler equations shown in equation 3.52, the elements of the Jacobian matrix A are not constant. Roe proposed replacing this non-linear equation with the linear equation

$$\frac{\partial \boldsymbol{Q}}{\partial t} + \bar{A}\frac{\partial \boldsymbol{Q}}{\partial x} = 0 \tag{3.72}$$

where $\bar{A}$ is a constant matrix. This equation is solved at the interface between two cells to determine the flux at the interface. The matrix $\bar{A}$ is chosen so that the solution of this linear equation gives the correct flux difference for the non-linear Riemann problem. The properties required of $\bar{A}$ are

i It constitutes a linear mapping from $\boldsymbol{Q}$ to $\boldsymbol{F}$

ii $\lim_{\boldsymbol{Q}^L \to \boldsymbol{Q}^R \to \boldsymbol{Q}} \bar{A}(\boldsymbol{Q}^L, \boldsymbol{Q}^R) = A(\boldsymbol{Q}) = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{Q}}$

iii $\boldsymbol{F}(\boldsymbol{Q}^R) - \boldsymbol{F}(\boldsymbol{Q}^L) = \bar{A}(\boldsymbol{Q}^L, \boldsymbol{Q}^R) \cdot (\boldsymbol{Q}^R - \boldsymbol{Q}^L)$

iv The eigenvectors of $\bar{A}$ are linearly independent

The superscript $()^L$ and $()^R$ represent quantities on the left and right sides of the interface.

The matrix $\bar{A}$ for the approximate Riemann problem is constructed from the flux Jacobian matrices where the primitive variables are replaced by the Roe averaged variables

$$\begin{aligned}
\bar{\rho} &= \sqrt{\rho^L \rho^R} \\
\bar{u} &= \frac{\sqrt{\rho^L} u^L + \sqrt{\rho^R} u^R}{\sqrt{\rho^L} + \sqrt{\rho^R}} \\
\bar{v} &= \frac{\sqrt{\rho^L} v^L + \sqrt{\rho^R} v^R}{\sqrt{\rho^L} + \sqrt{\rho^R}} \\
\bar{w} &= \frac{\sqrt{\rho^L} w^L + \sqrt{\rho^R} w^R}{\sqrt{\rho^L} + \sqrt{\rho^R}} \\
\bar{H} &= \frac{\sqrt{\rho^L} H^L + \sqrt{\rho^R} H^R}{\sqrt{\rho^L} + \sqrt{\rho^R}}
\end{aligned} \tag{3.73}$$

where H is the total enthalpy per unit mass, which is related to the total energy per unit mass by the relationship

$$H = E + \frac{p}{\rho} \tag{3.74}$$

The solution of the approximate Riemann problem yields the following equation for the flux at a cell interface

$$\boldsymbol{F}_{i+1/2,j,k} = \frac{1}{2}\left(\boldsymbol{F}^-_{i+1/2,j,k} + \boldsymbol{F}^+_{i+1/2,j,k}\right) - \frac{1}{2}\left|\bar{A}_{i+1/2,j,k}\right|\left(\boldsymbol{Q}_{i+1/2,j,k} - \boldsymbol{Q}_{i,j,k}\right) \tag{3.75}$$

where

$$\left|\bar{A}\right| = \bar{R}\left|\bar{\Lambda}\right|\bar{R}^{-1} \tag{3.76}$$

where the $(^-)$ notation is used to denote that the Roe averaged variables are used in the evaluation. The assumption is made that the waves from the solution of the one-dimensional Riemann problem moves normal to the interface. For the three-dimensional problem, the one-dimensional solution is repeated for the three directions. For first order spatial accuracy, the primitive variables used in the Roe averaged

variables come from the cells neighboring the interface. For second order accuracy, the values are extrapolated as shown in equation 3.71.

Newton Relaxation

Newton's method for a non-linear system of vector functions

$$\mathcal{F}(\boldsymbol{x}) = 0 \tag{3.77}$$

can be written as

$$\mathcal{F}'(\boldsymbol{x})\left(\boldsymbol{x}^{m+1} - \boldsymbol{x}^{m}\right) = -\mathcal{F}(\boldsymbol{x}^{m}) \tag{3.78}$$

This defines an iterative procedure for which m is the iteration counter and $\mathcal{F}'(\boldsymbol{x})$ is the Jacobian matrix defined by

$$\mathcal{F}'_{ij}(\boldsymbol{x}) = \frac{\partial \mathcal{F}_i(\boldsymbol{x})}{\partial \boldsymbol{x}_j} \tag{3.79}$$

Following the presentation of Whitfield [51], the discretized governing equation 3.59 leads to the function

$$\begin{aligned}\mathcal{F}(\boldsymbol{Q}^{n+1}) &= \frac{\Delta \boldsymbol{Q}^{n+1}}{\Delta\tau} + \delta_\xi \boldsymbol{F}^{n+1} + \delta_\eta \boldsymbol{G}^{n+1} + \delta_\zeta \boldsymbol{H}^{n+1} \\ &= \frac{\Delta \boldsymbol{Q}^{n+1}}{\Delta\tau} + \boldsymbol{R}(\boldsymbol{Q}^{n+1})\end{aligned} \tag{3.80}$$

for which a solution is sought by Newton's method. Here, the vector $\boldsymbol{Q}$ contains the dependent variables for every cell throughout the entire grid system. The Jacobian matrix is defined by

$$\mathcal{F}'_{ij}(\boldsymbol{Q}^{n+1}) = \frac{1}{\Delta\tau} + \frac{\partial \boldsymbol{R}(\boldsymbol{Q}^{n+1})}{\partial \boldsymbol{Q}^{n+1}} \tag{3.81}$$

which yields the iterative scheme

$$\left[\frac{1}{\Delta\tau_l} + \left(\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{Q}}\right)^{n+1,m}\right] \Delta \boldsymbol{Q}^{n+1,m+1} = -\left[\frac{\boldsymbol{Q}^{n+1,m} - \boldsymbol{Q}^{n}}{\Delta\tau_{min}} + \boldsymbol{R}(\boldsymbol{Q}^{n+1,m})\right] \tag{3.82}$$

where

$$\boldsymbol{Q}^{n+1,m+1} = \boldsymbol{Q}^{n+1,m} + \Delta\boldsymbol{Q}^{n+1,m+1}$$

$$\boldsymbol{Q}^{n+1,0} = \boldsymbol{Q}^{n}$$

n denotes the time level, m is the Newton iteration counter, $\Delta\tau_l$ is the local time step, and $\Delta\tau_{min}$ is the minimum time step. Flux vector splitting is used on the left-hand-side and flux difference splitting with Roe averaged variables is used on the right-hand-side. Steger-Warming jacobians, Roe analytical jacobians, or Roe numerical jacobians can be used.

Each iteration of the Newton's method is solved using symmetric Gauss-Seidel (SGS) iteration. The SGS iterations, or *inner iterations*, are performed on a grid by grid basis; while the Newton iterations, or *dt iterations*, are used to achieve time accuracy and are performed on all grids in sequence. This procedure eliminates synchronization errors at blocked and overset boundaries by iteratively bringing all dependent variables up to the τ^{n+1} time level. The fixed time step, $\Delta\tau_{min}$, is used to maintain time accuracy and a local time step, $\Delta\tau_l$, is used for stability and convergence of the Newton iterations. Steady state calculations do not use Newton iterations. The first term on the right-hand-side of equation 3.82 becomes zero and local time stepping is used during the inner iterations.

Explicit boundary conditions (BC) can be used or implicit BC's can be achieved by updating the BC's during the SGS relaxation solution of Equation 3.82 [52]. An under-relaxation factor is applied to the implicit BC update to improve stability.

Fixed-Point Iteration

A linear system of equations of the form

$$A\boldsymbol{x} = \boldsymbol{b} \tag{3.83}$$

can be solved using the general fixed-point iteration scheme

$$\boldsymbol{x}^{m+1} = \boldsymbol{x}^m + C\left(\boldsymbol{b} - A\boldsymbol{x}^m\right) \qquad m = 1, 2, 3, \ldots \tag{3.84}$$

(see, for example, Conte and de Boor [53, pages 223-233]). This iteration function is in quasi-Newton form. The function for which a zero is being sought is $\boldsymbol{f}(\boldsymbol{x}) = A\boldsymbol{x} - \boldsymbol{b}$. However, the matrix C is an approximate inverse of A rather than the inverse of the derivative of $\boldsymbol{f}$. This approximate inverse is defined by the requirement that

$$\|I - CA\| < 1 \tag{3.85}$$

for some matrix norm.

The coefficient matrix A can be written as

$$A = L + D + U \tag{3.86}$$

where L is the lower triangluar elements of A, D is the diagonal elements of A, and U is the upper triangluar elements of A. If A is diagonally dominant, D^{-1} is an approximate inverse of A and the iteration function

$$\boldsymbol{x}^{m+1} = \boldsymbol{x}^m + D^{-1}\left(\boldsymbol{b} - A\boldsymbol{x}^m\right) \tag{3.87}$$

will converge. This is Jacobi iteration. It can be rewritten as

$$D\boldsymbol{x}^{m+1} = \boldsymbol{b} - \left(L + U\right)\boldsymbol{x}^m \tag{3.88}$$

or as

$$x_i^{m+1} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^m - \sum_{j=i+1}^{n} a_{ij} x_j^m\right) \frac{1}{a_{ii}} \qquad i = 1, \ldots, n \tag{3.89}$$

to explicitly show how each element of $\boldsymbol{x}$ is updated. The distinguishing characteristic of Jacobi iteration is that the iteration function only uses values of $\boldsymbol{x}$ from the previous iteration.

Gauss-Seidel iteration comes from the choice of $C = (L + D)^{-1}$. This gives the iteration function

$$\boldsymbol{x}^{m+1} = \boldsymbol{x}^m + (L + D)^{-1}\left(\boldsymbol{b} - A\boldsymbol{x}^m\right) \tag{3.90}$$

This can be rewritten as

$$(L + D)\,\boldsymbol{x}^{m+1} = \boldsymbol{b} - U\boldsymbol{x}^{m} \tag{3.91}$$

or

$$D\boldsymbol{x}^{m+1} = \boldsymbol{b} - L\boldsymbol{x}^{m+1} - U\boldsymbol{x}^{m} \tag{3.92}$$

To explicitly show how each element is computed, this is written as

$$x_i^{m+1} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{m+1} - \sum_{j=i+1}^{n} a_{ij} x_j^{m} \right) \frac{1}{a_{ii}} \qquad i = 1, \ldots, n \tag{3.93}$$

As each element of $\boldsymbol{x}$ is updated, the previous elements of $\boldsymbol{x}$, which multiply the lower triangular elements of A, have already been updated. Thus, for the summation that represents $-L\boldsymbol{x}$, the elements of $\boldsymbol{x}$ are evaluated at iteration $m + 1$. In other words, when updating an element of $\boldsymbol{x}$, the most up to date values of $\boldsymbol{x}$ are used.

If Gauss-Seidel iteration is guaranteed to converge, it will converge faster than Jacobi iteration. It also has the side benefit that only one array is needed to store $\boldsymbol{x}$ during the iterations.

Parallel Considerations

Following the analysis presented by Weeratunga and Chawla [37], the solution algorithm could be written as a global system of linear equations

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,N} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N,1} & A_{N,2} & \cdots & A_{N,N} \end{bmatrix} \begin{Bmatrix} \Delta Q_1 \\ \Delta Q_2 \\ \vdots \\ \Delta Q_N \end{Bmatrix} = \begin{Bmatrix} -F_1(Q_1, Q_2, \cdots, Q_N) \\ -F_2(Q_1, Q_2, \cdots, Q_N) \\ \vdots \\ -F_N(Q_1, Q_2, \cdots, Q_N) \end{Bmatrix} \tag{3.94}$$

The diagonal, block matrix elements, A_{ii}, represent the coupling within grid i due to the implicit time discretization. These elements are banded, sparse matrices defined by the spatial discretization. The off-diagonal, block matrix elements, $A_{i,j} (i \neq j)$,

represent the coupling between grids i and j due to block-to-block and/or overlapping boundary conditions. The coupling between grids is dependent on the relative positions of the grids. Thus, some of the off diagonal elements will be zero.

Together, these elements form a large, sparse matrix. This large system of linear equations could be solved directly; however, this would not be efficient and does not lend itself well to parallel computing. Instead, the off-diagonal terms are moved to the right-hand-side. Thus, block-to-block and overlapped boundary conditions are treated explicitly. This gives a decoupled set of equations of the form

$$\begin{bmatrix} A_{1,1} & 0 & \cdots & 0 \\ 0 & A_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{N,N} \end{bmatrix} \begin{Bmatrix} \Delta \boldsymbol{Q}_1 \\ \Delta \boldsymbol{Q}_2 \\ \vdots \\ \Delta \boldsymbol{Q}_N \end{Bmatrix} = \begin{Bmatrix} -R_1 \\ -R_2 \\ \vdots \\ -R_N \end{Bmatrix} \tag{3.95}$$

where $-R_i = -F_i(\boldsymbol{Q}_1, \boldsymbol{Q}_2, \ldots) - \sum_{j \neq i} A_{i,j} \Delta \boldsymbol{Q}_j$. Each decoupled equation can be solved using Gauss-Seidel iteration.

CHAPTER 4
6DOF INTEGRATION

In order to solve store separation problems, we must be able to simulate the general motion of bodies under the influence of aerodynamic, gravitational, and externally applied loads. We will ignore structural bending; therefore, we can limit ourselves to rigid body motion. This chapter presents the basis for the six degrees of freedom (6DOF) motion simulation routines in Beggar that were written by Belk [4]. This is similar to the method presented by Meakin [54]. The equations of motion, the coordinate systems used, and the techniques used to integrate the equations of motion are presented.

Equations of Motion

The unconstrained motion of a rigid body is modeled by Newton's second law of motion

$$\boldsymbol{F} = m\boldsymbol{a} \tag{4.1}$$

where $\boldsymbol{F}$ is the total force acting on the body, m is the mass of the body, and $\boldsymbol{a}$ is the resulting acceleration of the body. This can be written as the conservation of linear and angular momentum

$$\boldsymbol{F} = \dot{\boldsymbol{L}} \tag{4.2}$$

$$\boldsymbol{M} = \dot{\boldsymbol{H}} \tag{4.3}$$

where $\boldsymbol{L} = m\boldsymbol{V}$ is the linear momentum, $\boldsymbol{H} = I\boldsymbol{\omega}$ is the angular momentum, and $\boldsymbol{M}$ is the total applied moments about the body center of gravity (CG). The dot notation

represents the derivative with respect to time, $\boldsymbol{V}$ is the translational velocity vector, $\boldsymbol{\omega}$ is the rotational velocity vector, and I is the rotational inertia tensor

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix} \tag{4.4}$$

constructed from the moments (I_{xx}, I_{yy}, I_{zz}) and products (I_{xy}, I_{xz}, I_{yz}) of inertia of the body. The six degrees of freedom of the motion are represented by the translational position of the CG and the rotation of the body about its CG.

Equations 4.2 and 4.3 can only be applied in an inertial reference frame (IRF); therefore, the derivatives of the linear and angular momentum must be taken with respect to an IRF. However, the body moments of inertia and products of inertia will vary with time (due to body motion) if they are defined relative to a fixed, global coordinate system. Thus, it is easier to use a non-inertial, local coordinate system that is fixed relative to the body, so that the moments and products of inertia will remain constant.

In order to apply equations 4.2 and 4.3 in a moving, local coordinate system, we need to relate the derivatives with respect to this non-inertial reference frame to derivatives with respect to an IRF. This relationship is defined by the equation

$$\dot{\boldsymbol{a}}\big|_{XYZ} = \dot{\boldsymbol{a}}\big|_{xyz} + \boldsymbol{\omega} \times \boldsymbol{a} \tag{4.5}$$

for any vector $\boldsymbol{a}$ defined in a coordinate system xyz that is rotating by $\boldsymbol{\omega}$ relative to an IRF XYZ. Applying this relationship to $\dot{\boldsymbol{L}}$ and assuming that the mass m is constant, equation 4.2 becomes

$$\frac{\boldsymbol{F}}{m} = \dot{\boldsymbol{V}}\Big|_{xyz} + \boldsymbol{\omega} \times \boldsymbol{V} \tag{4.6}$$

or

$$\dot{\boldsymbol{V}}\Big|_{xyz} = \frac{\boldsymbol{F}}{m} - \boldsymbol{\omega} \times \boldsymbol{V} \tag{4.7}$$

Applying 4.5 to $\dot{\boldsymbol{H}}$, equation 4.3 becomes

$$\boldsymbol{M} = I\dot{\boldsymbol{\omega}}|_{xyz} + \boldsymbol{\omega} \times \boldsymbol{H} \tag{4.8}$$

or

$$\dot{\boldsymbol{\omega}}|_{xyz} = I^{-1}\boldsymbol{M} - I^{-1}\boldsymbol{\omega} \times I\boldsymbol{\omega} \tag{4.9}$$

Equations 4.7 and 4.9 are the equations of motion written with respect to the local coordinate system (see Etkin [55] for a more complete derivation of the equations of motion). These equations can be integrated twice with respect to time to get a time history of the translational and rotational position of the rigid body. However, since the equations of motion are written with respect to the local coordinate system, the change in position coming from the integration of the equations of motion is of little use for tracking the body motion, since the local coordinate system is always changing. Instead, the changes in body position must be transformed to the global coordinate system so that the position and orientation of the body relative to the global coordinate system can be maintained.

Coordinate Transformations

The local coordinate system is represented by the lower case letters xyz, while the global coordinate system is represented by the upper case letters XYZ, as shown in figure 4.1. The origin of the local coordinate system is placed at the CG of the body, the $+x$ axis extends forward along the longitudinal body axis, the $+y$ axis extends laterally along what would be an aircraft's right wing (from the pilot's perspective), and the $+z$ axis extends downward in the direction defined by the right-hand rule.

The rotation of the local coordinate system relative to the global coordinate system can be represented by the three Euler angles of yaw (ψ), pitch (θ), and roll (ϕ). As shown in figure 4.1, the local coordinate axes, which are initially aligned with the global coordinate axes, are first rotated by ψ about the Z axis to produce the $x'y'Z$

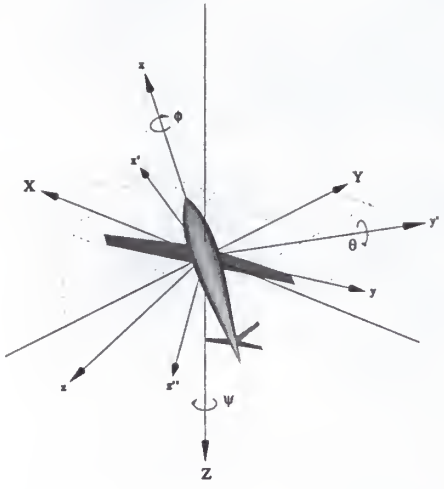


Figure 4.1: Transformation from global to local coordinates

axes. These axes are then rotated by θ about the y' axis to produce the $xy'z''$ axes. These axes are then rotated by ϕ about the x axis to produce the local coordinate axes xyz (see Blakelock [56] for another description of the coordinate systems). These transformations are written in matrix form as

$$\begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} x' \\ y' \\ Z \end{Bmatrix} = \begin{Bmatrix} X \\ Y \\ Z \end{Bmatrix} \tag{4.10}$$

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{Bmatrix} x \\ y' \\ z'' \end{Bmatrix} = \begin{Bmatrix} x' \\ y' \\ Z \end{Bmatrix} \tag{4.11}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = \begin{Bmatrix} x \\ y' \\ z'' \end{Bmatrix} \tag{4.12}$$

With the notation $[R_x(\phi)]$ representing a rotational transformation matrix constructed for rotation about the x axis by an angle ϕ, the complete transformation from local

coordinates to global coordinates can be written as

$$\left[R_z(\psi)\right]\left[R_y(\theta)\right]\left[R_x(\phi)\right]\begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = \begin{Bmatrix} X \\ Y \\ Z \end{Bmatrix} \tag{4.13}$$

or

$$\begin{bmatrix} \cos\psi\cos\theta & (\cos\psi\sin\theta\sin\phi- & (\cos\psi\sin\theta\cos\phi+ \\ & \sin\psi\cos\phi) & \sin\psi\sin\phi) \\ \sin\psi\cos\theta & (\sin\psi\sin\theta\sin\phi+ & (\sin\psi\sin\theta\cos\phi- \\ & \cos\psi\cos\phi) & \cos\psi\sin\phi) \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix}\begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = \begin{Bmatrix} X \\ Y \\ Z \end{Bmatrix} \tag{4.14}$$

Since the rotational transformations are orthonormal, the inverse transform is equal to the transpose. Thus, the complete transformation from global coordinates to local coordinates can be written as

$$\left[R_x(\phi)\right]^T\left[R_y(\theta)\right]^T\left[R_z(\psi)\right]^T\begin{Bmatrix} X \\ Y \\ Z \end{Bmatrix} = \begin{Bmatrix} x \\ y \\ z \end{Bmatrix} \tag{4.15}$$

which is equivalent to the transpose of the matrix shown in equation 4.14.

If the Euler angles ψ, θ, ϕ are used to track the angular position of the body relative to the global coordinate system, a relationship is required to convert the rotational velocity vector $\boldsymbol{\omega}$ in local coordinates (calculated from the integration of equation 4.9) to the rate of change of the Euler angles. However, the Euler angles are not applied about the global coordinate system axes; therefore, the transformation from local to global coordinates can not be used for this purpose. Referring back to figure 4.1, ψ is applied about the Z axis, θ is applied about the y' axis, and ϕ is applied about the x axis. Therefore, the rotational velocity vector can be decomposed as

$$\boldsymbol{\omega} = p\hat{\boldsymbol{e}}_x + q\hat{\boldsymbol{e}}_y + r\hat{\boldsymbol{e}}_z \tag{4.16}$$

or

$$\omega = \dot{\psi}\hat{e}_Z + \dot{\theta}\hat{e}_{y'} + \dot{\phi}\hat{e}_x \tag{4.17}$$

Decomposing the unit vectors $\hat{e}_{y'}$ and $\hat{e}_Z$ into the xyz coordinate system yields

$$\hat{e}_{y'} = \cos\phi\hat{e}_y - \sin\phi\hat{e}_z \tag{4.18}$$

and

$$\hat{e}_Z = -\sin\theta\hat{e}_x + \cos\theta\sin\phi\hat{e}_y + \cos\theta\cos\phi\hat{e}_z \tag{4.19}$$

as can be seen from the transformation matrices in equations 4.12 and 4.14. Combining equations 4.16–4.19 yields the relationships

$$\begin{aligned} p &= \dot{\phi} - \dot{\psi}\sin\theta \\ q &= \dot{\psi}\cos\theta\sin\phi + \dot{\theta}\cos\phi \\ r &= \dot{\psi}\cos\theta\cos\phi - \dot{\theta}\sin\phi \end{aligned} \tag{4.20}$$

which can be inverted to give

$$\begin{aligned} \dot{\phi} &= p + q\tan\theta\sin\phi + r\tan\theta\cos\phi \\ \dot{\theta} &= q\cos\phi - r\sin\phi \\ \dot{\psi} &= (q\sin\phi + r\cos\phi)/\cos\theta \end{aligned} \tag{4.21}$$

As $\theta \to \pi/2$, $\cos \to 0$ and $\tan \to \infty$; therefore, $\dot{\psi} \to \infty$ and $\dot{\phi} \to \infty$. This singularity is called "gimble lock" [57].

Quaternions

Quaternions were developed by Hamilton [58] as an extension to complex numbers and have been used in 6DOF simulations [59] because their use avoids the gimble lock problem. They have properties similar to both complex numbers and vectors.

Like a complex number, which has a real part and an imaginary part, a quaternion has a scalar part and a vector part and is often written as

$$\mathsf{Q} = e_0 + e_1\boldsymbol{i} + e_2\boldsymbol{j} + e_3\boldsymbol{k} \tag{4.22}$$

where $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$ are unit vectors in the three Cartesian coordinate directions.

The multiplication of two quaternions requires the additional rules of quaternion algebra

$$\begin{aligned} \boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 &= -1 \\ \boldsymbol{ij} = -\boldsymbol{ji} &= \boldsymbol{k} \\ \boldsymbol{jk} = -\boldsymbol{kj} &= \boldsymbol{i} \\ \boldsymbol{ki} = -\boldsymbol{ik} &= \boldsymbol{j} \end{aligned} \tag{4.23}$$

which are similar to the rules of complex math and vector cross products. The multiplication of two quaternions is simplified if we rewrite equation 4.22 as

$$\mathsf{Q} = Q_0 + \boldsymbol{Q} \tag{4.24}$$

which emphasizes the scalar and vector components. Following the distributive property, the multiplication of two quaternions is

$$\begin{aligned} \mathsf{PQ} &= (P_0 + \boldsymbol{P})(Q_0 + \boldsymbol{Q}) \\ &= P_0Q_0 + P_0\boldsymbol{Q} + Q_0\boldsymbol{P} + \boldsymbol{P} \otimes \boldsymbol{Q} \end{aligned} \tag{4.25}$$

The $\otimes$ operator can be shown to be equivalent to

$$\boldsymbol{P} \otimes \boldsymbol{Q} = \boldsymbol{P} \times \boldsymbol{Q} - \boldsymbol{P} \cdot \boldsymbol{Q} \tag{4.26}$$

Similar to complex arithmetic, the conjugate of a quaternion is defined as

$$\mathsf{Q}^* = Q_0 - \boldsymbol{Q} \tag{4.27}$$

The product of a quaternion and its conjugate is thus

$$\mathsf{QQ}^* = \mathsf{Q}^*\mathsf{Q} = e_0^2 + e_1^2 + e_2^2 + e_3^2 = |\mathsf{Q}|^2 \tag{4.28}$$

or the square of the magnitude of the quaternion. A unit quaternion is a quaternion of unit magnitude.

For the unit quaternion of the form

$$\mathsf{Q} = \cos(\alpha/2) + \boldsymbol{\lambda}\sin(\alpha/2) \tag{4.29}$$

the transformation

$$\mathsf{Q}\boldsymbol{V}\mathsf{Q}^* = \boldsymbol{V}' \tag{4.30}$$

rotates the vector $\boldsymbol{V}$ about the axis defined by the unit vector $\boldsymbol{\lambda}$ by an angle α to produce the vector $\boldsymbol{V}'$. Since this is a unit quaternion, Q^* is the inverse of Q. Thus the inverse transformation

$$\mathsf{Q}^*\boldsymbol{V}'\mathsf{Q} = \boldsymbol{V} \tag{4.31}$$

rotates the vector $\boldsymbol{V}'$ about the axis defined by $\boldsymbol{\lambda}$ by an angle of $-\alpha$ to recover $\boldsymbol{V}$.

If the unit vector $\boldsymbol{\lambda}$ is defined to be equivalent to $\hat{e}_x$ of our local coordinate system, α is equivalent to the roll angle ϕ and the rotational position of the body can be represented by the quaternion

$$\begin{aligned}\mathsf{Q} &= \cos(\phi/2) + \hat{e}_x \sin(\phi/2) \\ &= \cos(\phi/2) + [\cos\psi\cos\theta\boldsymbol{i} + \sin\psi\cos\theta\boldsymbol{j} - \sin\theta\boldsymbol{k}]\sin(\phi/2)\end{aligned} \tag{4.32}$$

where $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ represent the three cartesian coordinate directions $\hat{e}_X, \hat{e}_Y, \hat{e}_Z$ of the IRF. Then equation 4.30 represents the transformation from local coordinates to global coordinates and equation 4.31 represents the transformation from global coordinates to local coordinates. Equation 4.32 gives the relationship between the Euler angles and the components of the quaternion. Alternatively, the transformation in equation 4.31 can be compared to a general transformation matrix to find the relationship between the components of the quaternion and the elements of the transformation matrix.

The only other relationship needed in order to use quaternions to track rigid body motion is the knowledge of how to update the quaternion. Without going through a derivation, the following derivatives of the scalar and vector components of a quaternion were presented in Katz [60]

$$\dot{Q}_0 = -\frac{1}{2}\omega \cdot \boldsymbol{Q} \tag{4.33}$$

$$\dot{\boldsymbol{Q}} = \frac{1}{2}\omega Q_0 + \frac{1}{2}\omega \times \boldsymbol{Q} \tag{4.34}$$

These equations are integrated with respect to time along with the equations of motion.

The quaternion must remain a unit vector to ensure that the transformation represents pure rotation with no scaling or shearing. Therefore, the quaternion needs to be normalized during the integration.

Numerical Integration

A fourth order Runge-Kutta scheme is used to integrate the equations of motion. Runge-Kutta schemes are an attractive option for solving initial value problems governed by first order differential equations of the form

$$y' = f(x, y), \quad y(x_0) = y_0 \tag{4.35}$$

because they can achieve higher order accuracy without the evaluation of higher order derivatives. Conte and de Boor [53, pages 362-365] defines the fourth order Runge-Kutta scheme as

$$y_{n+1} = y_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{4.36}$$

where

$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_1)$$
$$k_3 = hf(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_2)$$
$$k_4 = hf(x_n + h, y_n + k_3)$$

The integration time step is represented by h, x represents the time, y represents the position, velocity, and quaternion, $f(x, y)$ represents the derivative of y (right hand side of the equations of motion), and the subscripts n and $n + 1$ are used to denote quantities at the current and next iteration (or time step), respectively.

The aerodynamics solution comes into the equations of motion through the integrated forces and moments. Since the aerodynamics are a function of position, the use of four different positions in the evaluation of $f(x, y)$ in equation 4.36 requires the calculation of the flow solution four times for each integration of the 6DOF. However, this would be very expensive. Therefore, the aerodynamics are assumed to be constant over the complete time step and are evaluated only once.

Since the translational equation of motion is written relative to the local coordinate system, the integrated aerodynamic forces (and moments) will be independent of position. However, the gravitational force, which is constant in global coordinates, is not constant in local coordinates. Thus, care should be taken when decomposing the gravitational force into local coordinates with each step of the Runge-Kutta integration.

CHAPTER 5
PARALLEL PROGRAMMING

Computing power has increased many orders of magnitude over the last decade. This trend is expected to continue in the near future. However, the shift appears to be away from sequential processing and towards parallel processing. This chapter presents an overview of parallel computing hardware, the options and some considerations for programming parallel computers, some methods for judging and improving parallel performance, and the proposed approach taken in this work.

Hardware Overview

The performance gains that are being achieved with single processors is diminishing as they approach physical limitations such as the speed of light. With this in mind, VLSI design principles have been used to conclude that it is possible to increase potential computing power more cost effectively by utilizing multiple, slower, less expensive components rather than a single, faster, more costly component [61]. Therefore, the trend in computing hardware is towards multiple processors. Machines that utilize high performance vector processors are shipping with modest numbers of vector processors, and massively parallel processors (MPP) are utilizing existing, low cost RISC processors (for example, the Cray T3D which uses DEC Alpha processors or the IBM SP2 which uses the IBM RS6000 processors) in ever increasing numbers to achieve greater potential processing power.

Another trend, that is affecting the way computing is being done, is the increase in network transfer rates. This allows physically separated resources to be utilized for solving a single problem. Since many MPP's utilize the same processors found in

high end workstations, a group of workstations connected by a high speed network can be viewed as a distributed parallel computer with the main differences being in the speed of the inter-processor connections and the possible differences in processors and operating systems. This type of computing is often referred to as "cycle harvesting." This is due to the fact that networked computers, that are used for routine computing during business hours, often sit idle at night. These unused computing cycles can be harvested for scientific computing.

A relatively new form of parallel computing takes the use of commercially available off-the-shelf components to the extreme. Personal computers, based on Intel or compatible microprocessors, running a freely available UNIX clone operating system such as LINUX, are linked together using low cost ethernet networking. Such parallel computers are often referred to as Beowulf clusters [62]. Such a distributed computing environment can represent a sizeable computational resource with very low associated cost.

Parallel computer architectures are often classified according to the number of instructions that can be executed in parallel, as well as, the amount of data that can be operated on in parallel. The most common of these classifications range from multiple instruction, multiple data or MIMD computers to single instruction, multiple data or SIMD computers. SIMD systems offer reduced program complexity, but can greatly reduce the available algorithms than can be implemented on such an architecture. Parallel computers are often further classified according to their memory layout as distributed memory, in which case each processor has its own local memory, or as shared memory, for which each processor has direct access to a single, global memory address space. Most of the machines being produced today are of the MIMD type. The Cray T3D and IBM SP2 are distributed memory MIMD machines, while the SGI Onyx is a shared memory MIMD machine.

The SGI Origin 2000 represents a unique memory architecture referred to as CC-NUMA (cache-coherent, nonuniform memory access). It is made of multiple

node cards that contain two processors and local, shared memory. However, any processor on any node card can access any of the memory in the machine. This hybrid organization of memory is called distributed, shared memory (DSM). There is a latency associated with accessing memory located off of a node card; therefore, access times to memory are nonuniform. However, hardware is used to maintain coherency of data held in cache between different processors. This architecture has been shown to perform well for many irregular applications and scales well to moderate numbers of processors [63].

Software Overview

Logically, parallel computers can be viewed as a set of sequential processors, each with its own memory, inter-connected by some communication links [61]. Each processor executes a sequential set of instructions and communicates with other processors and accesses remote memory through the communication links. Distributed memory and shared memory systems, as well as, distributed computing environments fit this model. The processors of a shared memory system simply have a more efficient way of accessing remote memory than do the processors of a distributed memory system or a distributed computing environment. This model of a parallel computer and the use of messages for all communication between processors forms the basis of the *message passing* paradigm of parallel programming.

Due to the model used for the parallel computer, it is conceivable that the user could write, compile, and execute a different program on each processor, with each program communicating with the others via messages. It is more often the case that the same source is compiled and executed on each processor, with control flow statements in the code used to determine the path executed or the data manipulated at run time. This programming model is referred to as single process multiple data or SPMD. The SPMD model of programming aids in code maintenance and provides a simplified path for converting an existing sequential code for parallel execution.

Many libraries exist for implementing message passing. Two of the more predominant libraries are PVM [64] and MPI [65]. PVM, which stands for parallel virtual machine, is a defacto standard message passing interface due to its popularity and widespread use. It is the product of Oak Ridge National Lab and several university contributions. PVM consists of two parts: a library consisting of the functions that implement the application programming interface (API) and a daemon which runs in the background and actually handles the communication between processes. MPI, which stands for Message Passing Interface, is a proposed standard message passing interface. It was developed out of a series of meetings of a committee of experts from the parallel computing community. MPI draws features from other message passing libraries and provides a common API that the vendors can optimize for their machines. PVM evolved out of a research project on distributed computing and places a higher priority on portability than on performance. MPI is expected to provide better performance on large MPP's but does not provide for heterogeneous distributed computing and lacks many task management functions [66].

Other models are available for parallel programming. One of the more popular is the shared memory programming model. Pthreads [67] is a POSIX standard implementation for shared memory programming using *threads*. A thread is a light weight process that shares memory with other threads, but has its own program counter, registers, and stack so that each thread can execute a different part of a code. The sharing of memory between threads is automatic and communication between threads is accomplished through cooperative use of shared variables. Mutual exclusion or *mutex* variables are used to ensure that only one thread changes the value of a variable at a time. Signals are sent between threads using *condition* variables. OpenMP [68] is an alternative library that attempts to avoid the low level programming constructs required by Pthreads. OpenMP is used to identify loops that can be executed in parallel similar to vectorization of loops on vector processors. OpenMP automatically handles all communication.

These techniques all require shared memory and thus can not be used on distributed memory, parallel computers. However, Pthreads or OpenMP can be mixed with PVM or MPI to take advantage of both programming models when clusters of shared memory multi-processor (SMP) machines are linked together. Likewise, other techniques for using shared memory can be mixed with the message passing model. POSIX also defines a standard for specifying the use of shared memory explicitly [69] as opposed to the automatic use of shared memory as with Pthreads.

When approaching a parallel programming task, the key issues to be addressed are concurrency, scalability, locality, and modularity [61]. Concurrency relates to the need for algorithms which subdivide larger problems into a set of smaller tasks that can be executed concurrently. An intimate knowledge of the data structures and data dependencies in an algorithm is required to identify such concurrencies. Scalability relates to the behavior of an algorithm in terms of parallel efficiency or speedup as a function of processor count. Since the number of processors being utilized in MPP's appears to be continually increasing, the efficiency of a good parallel program design should scale with increased processor counts to remain effective throughout its life cycle. Locality relates to the desire to enhance local memory utilization since access to local memory is less expensive than access to remote memory. Raw communication speeds are typically orders of magnitude slower than floating-point operations; thus, communication performance strongly influences the parallel run time. Modularity is important in all software development. It allows objects to be manipulated without regard for their internal structure. It reduces code complexity and promotes code reuse, extensibility, and portability.

The algorithm design process can be broken down into four phases: partitioning, communication, agglomeration, and mapping [61]. Machine independent issues, such as concurrency, are considered early in the design process, while machine specific issues are delayed until late in the design. Partitioning and communication address the issues of concurrency and scalability, while agglomeration and mapping address

locality and performance. Partitioning falls into two major categories: functional decomposition and data decomposition. Functional decomposition focuses on the computation, while data decomposition focuses on the data. A good partition will divide both the data and the computation. The communication phase of a design deals with identifying the inter-process communication requirements. This is complicated when the communication patterns are global, unstructured, dynamic, and/or asynchronous. Agglomeration seeks to reduce communication costs by increasing computation and communication granularity. Tasks can be combined and data and/or computation can be duplicated across processors in order to reduce communication. The mapping phase is a machine specific problem of specifying where each task will execute. A mapping solution is highly dependent on the communication structure and the work load distribution. A load balancing algorithm is often needed. If the communication structure is dynamic, tradeoffs must be made between a load imbalance and repeated application of a possibly expensive load balancing algorithm.

A good algorithm design must optimize a problem-specific function of execution time, memory requirements, implementation costs, and maintenance costs, etc. [61]. Furthermore, when solving coupled systems of partial differential equations, issues unique to the problem must be considered. For example, on a distributed memory machine, a minimum number of processors may be required in order to hold a specific problem; however, the use of additional processors must be balanced against its effect on the solution convergence [70]. Likewise, since communication cost is proportional to surface area and computational cost is proportional to volume, the desire for a high ratio of volume to surface area places a lower limit on the subdivision of the computational domain. Communication through messages has an associated cost of the latency time for message startup and a cost per word of data transferred in the message; therefore, it is generally desirable to use a small number of larger messages rather than a large number of small messages. However, the use of small messages may allow an algorithm change that would allow communications to be overlapped

by computation. An efficient parallel implementation will require the consideration of all such factors.

Performance

Performance of a parallel algorithm is normally measured via *speedup*. This is the ratio of the execution time on a single processor and the execution time on multiple processors. Thus, the speedup s can be computed by

$$s = \frac{T_1}{T_n} \tag{5.1}$$

where T_1 denotes the execution time on a single processor and T_n denotes the execution time on n processors. Ideally, T_1 should represent the execution time of the best sequential algorithm available to do the job. When parallelizing a sequential algorithm, the *best* sequential algorithm may not parallelize well and, vice versa, the best parallel algorithm may not perform well sequentially. Likewise, when parallelizing a given sequential algorithm, some overhead will be introduced. If the parallel algorithm is executed on a single processor to measure T_1, this value may be artificially high due to the use of a poor sequential algorithm or due to the existence of parallelization overhead. However, the definition of the *best* sequential algorithm may be unattainable. Thus, there exists some ambiguity in how T_1 should be measured in order to judge the performance of a parallel algorithm. At the least, when converting an existing sequential algorithm for execution in parallel, T_1 should be measured using the original sequential algorithm. Likewise, if any algorithm changes are made during parallelization that would also decrease the sequential execution time, T_1 should be remeasured so as to isolate improvements due to the algorithm change from improvements due to the use of multiple processors.

One source of overhead, that exists in all parallel programs, is time spent in communication between multiple processors. Following the analysis presented by Roose and Van Driessche [71], the total execution time of a parallel algorithm executed

on n processors can be approximated as

$$T_n = T_{\text{calc}} + T_{\text{comm}} \tag{5.2}$$

where T_{calc} denotes the actual computation time and T_{comm} denotes the time spent in communication due to parallelization. If the work is perfectly balanced and there is no time spent in communication during a sequential run, then the execution time of the sequential run will be

$$T_1 = n\,T_{\text{calc}} \tag{5.3}$$

Hence, the speedup would be

$$\begin{aligned} s &= \frac{n\,T_{\text{calc}}}{T_{\text{calc}} + T_{\text{comm}}} \\ &= \frac{n}{1 + \frac{T_{\text{comm}}}{T_{\text{calc}}}} \end{aligned} \tag{5.4}$$

Thus, the ratio of the communication time and the computation time can have a large effect on the speedup.

In general, for CFD flow solvers, the communication time is proportional to the area of (number of grid points on) the boundaries of the domain, and the computation time is proportional to the volume of (total number of grid points in) the domain. Thus, as the problem size increases, the ratio of communication to computation decreases. The characteristics of a particular computer, the form of the communication, the algorithm used, and the partitioning of the domain can also affect this ratio.

In general, a parallel computer with n processors can execute n instructions at the same time. Thus, if the instructions in a sequential algorithm could be evenly divided among the n processors, so that each processor executed $1/n^{th}$ of the total instructions, the execution time would be decreased by a factor of n. Therefore, linear speedup is the ideal case, and speedup is limited to $s \leq n$. However, there are other factors that place additional limits on the speedup that can be achieved.

If we consider the entire work load of a complete simulation to be broken down into part that can be executed in parallel and part that must be executed serially,

the speedup, that can be achieved, is limited by Amdahl's law [72]:

$$s \leq \frac{1}{\left(\frac{f_p}{n}\right) + f_s} \tag{5.5}$$

where f_s is the serial fraction of the work, f_p is the parallel fraction of the work, and n is the number of processors on which the parallel portion of the code is running. The factors f_s and f_p are fractions so that

$$0 \leq f_s \leq 1$$

$$0 \leq f_p \leq 1$$

and

$$f_s + f_p = 1 \tag{5.6}$$

Since the parallel work will be distributed across multiple processors, the execution time of the parallel work will be decreased, but the execution time of the serial work will not.

Amdahl's law shows the significant penalty that the serial fraction of the work load can place on the parallel performance. For example, consider a case where 5% of an executable code must be performed serially ($f_s = .05$ and $f_p = .95$). If only 4 processors are used, the speedup will be limited to 3.48, nearly 90% of the ideal speedup. However, if 32 processors are used, then the speedup will be limited to 12.55, less than 40% of the ideal speedup. Although the processor count was increased by a factor of 8, the speedup increased by less than a factor of 4. In fact, as the number of processors $n \to \infty$, the term $f_p/n \to 0$. Thus, the speedup is limited to $1/f_s$, or 20 in this case, no matter how many processors are used.

This could be used to argue that parallel processing does not hold the answer to the need for increased computing power. However, the potential from multiple processors and the increased memory often available with MPP machines allows larger and larger problems to be addressed. With CFD solutions, as the problem size increases,

the computation to communication ratio usually increases and the serial fraction of the work load decreases.

Even the limit specified by Amdahl's law is not always reached. The major contributor to this behavior is an imbalance in the distribution of the work to be executed in parallel. Consider figure 5.1. The left side of the figure shows the serial execution of a function that operates on four grids, while the right side shows the parallel execution of the function on four processors with one grid per processor. The serial execution time, and thus the total work, is represented by the time T5-T1. On four processors, the average work per processor is represented by the time (T5-T1)/4. However, the total execution time in parallel is dictated by the maximum execution time of any process. This time, T2-T1, is larger that the average execution time by a factor related to the imbalance in the work load or execution times.

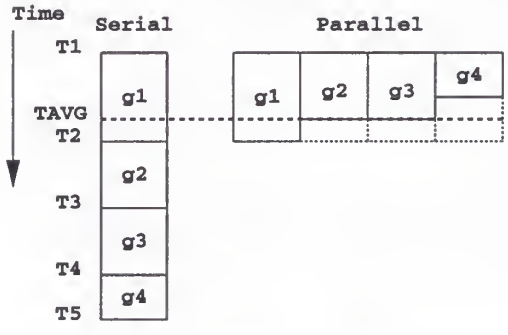


Figure 5.1: Unbalanced work load

Since the term f_p/n in equation 5.5 represents the average parallel work per processor, this work must be increased by a factor proportional to the load imbalance. Generalizing equation 5.5 to include the effect of a load imbalance, the speedup becomes

$$s \simeq \frac{1}{\left(\frac{f_p}{n}\right) * f_i + f_s} \tag{5.7}$$

where f_i is the load imbalance factor. The load imbalance is often judged by the ratio of the maximum execution time of any process and the minimum execution time of

any process. However, as used here, the load imbalance factor is used to increase the average execution time per process to the maximum execution time of any processor. Thus, the imbalance is equal to the ratio of the maximum execution time of any process to the average execution time per process.

The load balance is further complicated by the basis for the decomposition of the work. If each division in the decomposition does not represent a nearly equal piece of the work, the load balance can vary significantly with the process count. Obviously, if there are not enough pieces of work, some of the processes would sit idle. Likewise, if there is one piece of work that is significantly larger than the other pieces, it can dominate the execution time. Consider figure 5.2. The left side of the figure shows the serial execution of a function that operates on four grids. When the function is duplicated and the grids are distributed across two processes (shown in the middle of the figure), the work is well balanced and the execution time is cut in half. However, when four processes are used (shown on the right side of the figure), no improvement in the execution time is seen. The work associated with grid g1 is one half of the total work; thus, the execution time is dominated by the execution time of g1.

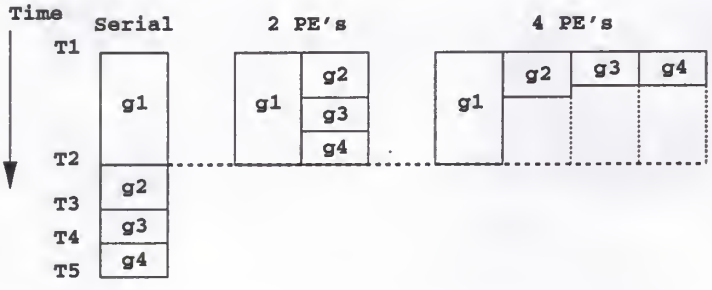


Figure 5.2: Limitations in load balance caused by a poor decomposition

Another common cause for the degredation in the speedup achieved is synchronization between processes. Synchronization is enforced by the placement of barriers in the execution path. No process may pass the barrier until all of the processes have reached the barrier. This can ensure that every process has completed a particular portion of the work before any process starts on the next portion of work. This may

be required if one function is dependent on the results from a previous function. How this can cause an increase in execution time is illustrated in figure 5.3. This diagram shows two functions (A and B) operating on separate grids on separate processors. Without synchronization, the total work per process may be well balanced; but if synchronization is required between the functions, wait time can be introduced if each function is not well balanced.

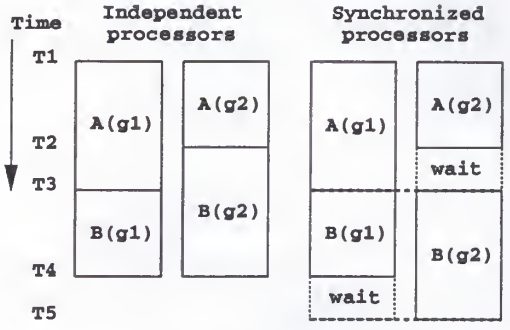


Figure 5.3: Imbalance caused by synchronization

This illustrates the fact that each piece of work between any two synchronization points must be well balanced in order to achieve a good overall load balance for the entire code. To take this into account, equation 5.7 should be written as

$$s \simeq \frac{1}{\sum \left(\frac{f_p}{n} * f_i \right) + f_s} \tag{5.8}$$

where the terms within the summation represent each section of code between synchronization points that is executed in parallel.

Load Balancing

The most important part to achieving good parallel performance is load balancing. The problem of load balancing is similar to the computer science problems referred to as the "knapsack problem" [73] and the "partition problem"[74]. These problems are NP-complete, which is the set of all problems for which no algorithm exists that is guaranteed to produce the exact solution through nondeterministic means

in polynomial time. The input to the knapsack problem is defined by a set of items with specified sizes and values and a knapsack of a specified capacity. The problem is to maximize the value of the subset of items that will fit into the knapsack at one time. The input to the partition problem is a set of blocks of varying heights. The problem is to stack the blocks into two towers of equal heights.

The input to the load balancing problem consists of a set of pieces of work, a measure of the cost of each piece of work, and the number of processors across which the pieces of work are to be distributed. The problem is to associate each piece of work with a processor, while minimizing the ratio of the maximum total work associated with any processor and the average work per processor. The amount of work associated with each piece of work is similar to the value of the items to be placed in the knapsack or the height of the blocks. The processors are similar to the knapsack or the towers. The average work per processor corresponds to the capacity of the knapsack or the average height of the towers. However, each piece of work must be associated with a processor, each processor must have at least one piece of work, and there is no limit on the amount of work that can be associated with each processor.

The algorithm used to balance the work of the flow solver is a max-min algorithm. This algorithm, shown below, takes the piece of work with the maximum cost from the pieces of work that have not yet been assigned to a processor and assigns it to the processor that has the minimum total work assigned to it. This algorithm distributes the work across the available processors with only a single pass through the list of the pieces of work, thus the execution time is bounded. With sufficient partitioning of the work, this algorithm produces a good load balance, although it may not produce the best possible distribution of the work.

The array $Work[]$ is an estimate of the cost associated with each piece of work (each grid, in this case). Since the work of the flow solver is closely associated with the number of grid points, the cost associated with a grid can be defined as the number

of points in the grid. However, there are other factors that can affect the execution time that are not directly related to the number of grid points. Thus, a user defined weight can be associated with each grid and it is the weighted number of grid points that is fed into the load balancing routine as the cost of each piece of work. The output from the load balancing routine is an array $GridToPe[]$ that specifies which processor will execute the flow solver for each grid.

```
MAP_GRIDS_TO_PES(Work[])
 1  for i ← 1 to npes
 2  do PeWork[i] ← 0
 3
 4  for i ← 1 to ngrids
 5  do PeNum ← FIND_MIN_VAL_INDEX(PeWork[])
 6     GridNum ← FIND_MAX_VAL_INDEX(Work[])
 7     PeWork[PeNum] ← PeWork[PeNum] + Work[GridNum]
 8     GridToPe[GridNum] ← PeNum
 9     Work[GridNum] ← 0
10
11  return GridToPe[]
```

This load balancing algorithm is applied to the flow solver, for which there is an initial estimate of the value of each piece of work. In grid assembly, there is no apriori knowledge of the amount of work associated with a grid or superblock. In fact, the amount of work associated with a grid or superblock depends on the location of the grids and thus changes as the grids move. In such a case, a dynamic load balancing algorithm is needed that can redistribute the work based on some perception of the current work distribution.

The algorithm implemented for dynamic load balancing, shown below, consists of two iterative steps. The algorithm requires as input, some numerical measure of the cost associated with each piece of work in the partition and the current mapping of those pieces of work to the available processes. The first step in the algorithm is to move work from the process with the maximum work load to the process with the minimum work load in order to improve the load balance. The second step is to swap single pieces of work between two processes in order to improve the load balance.

The output from this algorithm is a new mapping of the pieces of work to the set of processes. This mapping must be compared to the previous mapping to determine which data has to be transferred from one process to another.

```
OPTIMIZE_MAPPING(Work[], WorkToPe[])
 1  TotalWork ← CALC_SUM(Work[])
 2  AvgWork ← TotalWork/npes
 3  PeWork[], PeWorkList[] ← BUILD_PE_WORK_LISTS(WorkToPe[])
 4
 5  MOVE_WORK(Work[], WorkToPe[])
 6  SWAP_WORK(Work[], WorkToPe[])


MOVE_WORK(Work[], WorkToPe[])
 1  repeat
 2          pemin ← FIND_MIN_VAL_INDEX(PeWork[])
 3          pemax ← FIND_MAX_VAL_INDEX(PeWork[])
 4
 5          WorkLimit ← (PeWork[pemax] − PeWork[pemin]) * 0.99
 6          WorkToPut ← CHOOSE_MAX_LIMITED_VAL(
 7               PeWorkList[pemax], WorkLimit)
 8
 9          if WorkToPut
10            then WorkToPe[WorkToPut] ← pemin
11                 PeWork[], PeWorkList[] ← BUILD_PE_WORK_LISTS(
12                      WorkToPe[])
13    until WorkToPut = NIL
```

In line 3 of OPTIMIZE_MAPPING, BUILD_PE_WORK_LISTS calculates the total work per process ($PeWork[]$) and also builds an array of the lists of pieces of work that are mapped to each process ($PeWorkList[]$) from the mapping of work to processes ($WorkToPe[]$). In MOVE_WORK, if any piece of work can be moved from one process to another and decrease the maximum amount of work on any process, then it will improve the load balance. Therefore, in line 5, $WorkLimit$ is set based on a percentage of the difference in the work assigned to the processes with the least and most work. CHOOSE_MAX_LIMITED_VAL chooses the piece of work from the list of work associated with $pemax$ ($PeWorkList[pemax]$) that has the largest cost and also is less than $WorkLimit$. This piece of work is assigned to $pemin$ in line 10 and the

work lists are updated in line 11.

```
SWAP_WORK(Work[], WorkToPe[])
 1  repeat
 2          for i ← 1 to npes
 3          do PeWorkImb[i] ← PeWork[i] − AvgWork
 4          pemax ← FIND_MAX_VAL_INDEX(PeWorkImb[])
 5
 6          for  each i in PeWorkList[pemax]
 7          do WorkToPut ← i
 8             for pemin ← 1 to npes
 9             do if pemin ≠ pemax
10                   then WorkLimit ← (Work[WorkToPut]
11                              −PeWorkImb[pemax]
12                              +PeWorkImb[j]) * 1.001
13
14                        WorkToGet ← CHOOSE_ANY_LIMITED_VAL(
15                           PeWorkList[pemin])
16
17                        if WorkToGet
18                          then WorkToPe[WorkToGet] ← pemax
19                               WorkToPe[WorkToPut] ← pemin
20                               PeWork[], PeWorkList[] ←
21                                   BUILD_PE_WORK_LISTS(
22                                      WorkToPe[])
23                               break
24
25    until WorkToGet = NIL
```

In SWAP_WORK, one piece of work on one process is swapped for a piece of work on another process. Therefore, there is an upper and lower bound on the cost of the *WorkToGet*. If *WorkToGet* is larger that *WorkToPut*, then the total work on *pemax* will increase. However, if *WorkToGet* is less than *WorkToPut* by more than the difference between the imbalance on the two processes, then the imbalance on *pemin* will increase beyond the original imbalance of *pemax*. The routine CHOOSE_ANY_LIMITED_VAL chooses a piece of work from the list of work associated with *pemin* (*PeWorkList*[*pemin*]) that costs less than the work represented by *WorkToPut* and is greater than *WorkLimit*. It is not important that the optimum piece of work be chosen. Any piece of work that improves the load imbalance

will do. The work represented by *WorkToGet* is assigned to *pemax* and the work represented by *WorkToPut* is assigned to *pemin*. The work lists are updated by BUILD_PE_WORK_LISTS. The **break** at line 23 causes control to jump out of the loop that started at line 6 so that the load imbalances and *pemax* can be recalculated.

Each step of this algorithm attempts to decrease the load imbalance caused by the process with the maximum work load. However, the search for pieces of work to swap is exhaustive. This algorithm is used to redistribute the superblocks based on the work of grid assembly; therefore, this algorithm is not too expensive, since there are not many superblocks. This algorithm could be extended to swap multiple pieces of work on one process for one or more pieces of work on another process. However, this would require more efficient ways of sorting the pieces of work, so that the pieces to be swapped could be selected more efficiently.

Another situation, that often arises, is a large set of pieces of work that can be treated as an array. This array can be evenly divided among the processes with each getting an equal number of elements of the array. However, if each element of the array does not represent the same amount of work, there will be a load imbalance. It could be expensive to treat each element as a separate piece of work, measure its cost, and use the previous algorithms to distribute this large number of pieces of work. Instead, the total cost of the work can be associated with the processor and used as a weight. Load balancing can then be achieved by dividing the array so that the weighted number of elements is equally divided among the processes.

This algorithm requires as input, the number of elements of the array mapped to each process ($N[]$) and the execution time of each process ($T[]$). A weight for each process ($W[]$) is calculated as the execution time per array element. The excess number of elements assigned to the process with the maximum load is calculated as the delta between the process execution time and the average process execution time divided by the process weight. Since this number of elements will be assigned to another process, the weight of the receiving process must be updated. The execution

times of the two processes are updated and the loop is repeated if there are other processes with excess array elements.

```
OPTIMIZE_ARRAY_MAPPING(N[], T[])
 1  for i ← 1 to npes
 2  do W[i] ← T[i]/N[i]
 3  imin ← FIND_MIN_VAL_INDEX(T[])
 4  imax ← FIND_MAX_VAL_INDEX(T[])
 5  Tavg ← CALCULATE_AVG_VAL(T[])
 6
 7  repeat
 8        Nexcess ← (T[imax] − Tavg)/W[imax]
 9        if Nexcess = 0
10          then break
11        TotWeight ← N[imin] * W[imin] + Nexcess * W[imax]
12        W[imin] ← TotWeight/(N[imin] + Nexcess)
13        N[imin] ← N[imin] + Nexcess
14        N[imax] ← N[imax] − Nexcess
15        T[imin] ← N[imin] * W[imin]
16        T[imax] ← N[imax] * W[imax]
17        imin ← FIND_MIN_VAL_INDEX(T[])
18        imax ← FIND_MAX_VAL_INDEX(T[])
19   until T[imax]/Tavg < 1.005
```

Proposed Approach

Since this work builds on the initial parallel implementation of the Beggar flow solver [6], the same methods used there will be continued here. The message passing paradigm is used within an SPMD programming model. PVM is used for the message passing environment. The code is geared toward MIMD machines with either distributed or shared memory. The ultimate goal is to allow heterogeneous computing although homogeneous computing environments are the primary focus. A functional decomposition of the entire simulation process is used with the flow solution, force and moment calculation, 6DOF integration, and grid assembly being the primary functions. Coarse grain domain decomposition of the flow solver based on grids is used. The degree to which domain decomposition of the grid assembly function can be used is determined. Load balancing is treated as the primary contributor to good

parallel performance. The most efficient implementation requires consideration of both the flow solver and the grid assembly process during data partitioning and load balancing.

For parallel algorithm design, Foster [61] recommends a process denoted by the acronym PCAM, referring to partitioning, communication, agglomeration, and mapping, as mentioned previously. In this approach, Foster recommends that the finest grained partitioning of the work be defined along with all of the required communications. Then, the partitions are agglomerated to reduce the communications required and thus increase the computation to communication ratio. The final step is to map the work to processors based on the particular computer architecture. In this work, a nearly opposite approach is taken. The work is partitioned using coarse grain decomposition first. This allows a parallel implementation to be achieved with minimal code changes and with less expertise in the existing sequential code, as well as, parallel programming itself. This also allows the users to receive and to start using the code earlier. As the code performance is analyzed, the granularity of the decomposition is refined as required. Mapping of work to processes is done dynamically to achieve a good load balance; however, no machine specific issues of mapping work to specific processors are addressed.

CHAPTER 6
PARALLEL IMPLEMENTATIONS

Phase I: Hybrid Parallel-Sequential

The simplest approach to achieving a parallel version of Beggar for moving body problems is to use a separate front-end (FE) process that performs the grid assembly function for the complete domain in a serial fashion with respect to the parallel execution of the flow solver across multiple back-end (BE) processes. This requires that proper communication be established between the flow solution function and the grid assembly function; however, this does not require any consideration of load balancing or partitioning of the grid assembly function.

This implementation is referred to as phase I and is depicted in figure 6.1. This diagram and the others like it that follow are referred to as timing diagrams. The major functions are represented and the diagram serves as a template of one iteration of the solution process. The vertical direction represents time and this template can be stamped out in a vertical manner to construct a complete time history of the solution process. The boxes on the left represent the functions running in the FE process, while the boxes on the right represent the functions running in the BE processes. The arrows represent communication between specific functions on the FE and BE. Communication between functions on the same process is not shown explicitly. The vertical lines through a function indicates that it is spread across multiple processors. Although these diagrams are not drawn to scale, the work of a particular function is represented by the area of the box drawn for that function. Thus, as a function is spread across multiple processors, the width increases and the height decreases representing the decrease in the time for executing the function.

Referring to figure 6.1, the solution process is started at time T1. Once the grid assembly function is completed at time T2, the interpolation stencils, iblank arrays, etc. are communicated from the FE to the BE so that the flow solution function can proceed. Once an iteration of the flow solver is completed, the forces and moments are integrated and are passed from the BE to the FE. The 6DOF function is then executed to reposition the grids and to calculate motion rates. Since the 6DOF function executes quickly, it is duplicated on the FE and the BE rather than communicating the resulting information.

Ignoring the cost of the force and moment calculation and the 6DOF integration, the flow solver represents the parallel work and the grid assembly represents the serial work. Based on the fractions of the total execution time represented by the flow solver and the grid assembly, equation 5.7 can be used to estimate the performance that can be achieved with this implementation. However, instead of using the notation f_p, f_i and f_s, we will use the uppercase letters F and G to represent the flow solver and grid assembly functions and the subscripts p, s and i to represent the fractions of the work executed in parallel or serial and the load imbalance factors, respectively. Thus, for the phase I implementation, the speedup can be approximated as

$$s \simeq \frac{1}{\left(\frac{F_p}{nbes}\right) * F_i + G_s} \tag{6.1}$$

where *nbes* is the number of BE processes. Since the work of the flow solver is closely

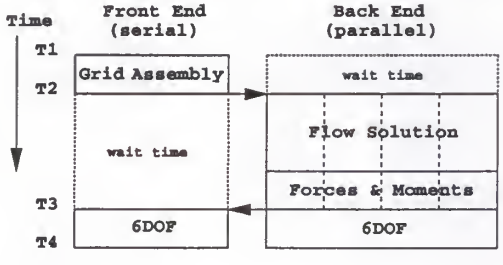


Figure 6.1: Phase I implementation

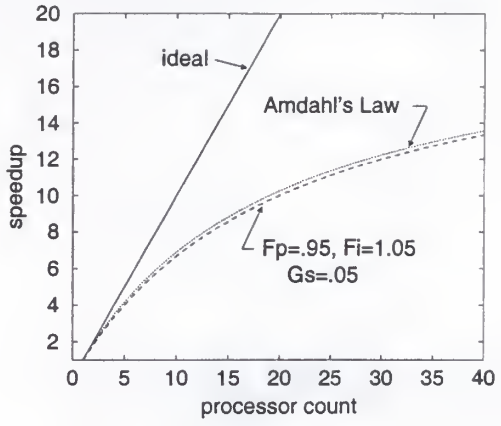


Figure 6.2: Comparison of estimated speedup of phase I to Amdahl's law

related to the number of grid points, the equation

$$F_i = \frac{max(\text{no. points on each processor})}{\text{avg no. points per processor}} \tag{6.2}$$

can be used to obtain an approximation for the load imbalance factor for the flow solver. There are other factors, such as boundary conditions and the distribution of communication between processors, that affect the load balance. They will be ignored at this point.

Figure 6.2 shows a comparison of the estimated speedup from the phase I implementation to Amdahl's law. The estimated speedup of phase I is plotted using equation 6.1 with $F_p = 0.95$, $G_s = 0.05$, and $F_i = 1.05$ representing a nominal load imbalance of 5% in the distribution of the work of the flow solver. This plot shows the significant drop off in speedup with increased processor counts due to the serial fraction of the work. A small decrease in the performance of the phase I implementation (as compared to Amdahl's Law), due to the load imbalance, can also be seen.

Phase II: Function Overlapping

Some parallel efficiency can be gained by overlapping the grid assembly function with the flow solution function. This overlapping could be achieved by updating the

6DOF and the interpolation stencils at the same time that the flow solver is updated, by using the forces and moments calculated from a previous iteration of the flow solution as an approximation to the current forces and moments. Thus, the updating of the grid assembly would be based on a flow solution that is lagged behind the current flow solution. This is similar to the technique used for sequential processing in Lijewski and Suhs [28, 29], where store separation events were simulated with the grid assembly recomputed once after every 20 iterations of the flow solver and 6DOF integration. The grids were moved and the grid motion time metrics were fed into the flow solver every iteration, although the interpolation stencils were not. Time accurate forces and moments were used, although the flow solution could be affected since the interpolation stencil locations were not time accurate. The variation in stencil locations due to this time lag (.004 seconds in their case) is justified by the assumption that the grids will not move by an appreciable amount during the delay. Good results were achieved for the problems addressed.

Some parallel efficiency may be gained without lagging the grid assembly behind the flow solution. This is possible due to the Newton-Relaxation scheme used in the flow solution function. The discretized, linearized, governing equations are written in the form of Newton's method. Each step of the Newton's method is solved using symmetric Gauss-Seidel (SGS) iteration. The SGS iterations, or *inner iterations*, are performed on a grid by grid basis, while the Newton iterations, or *dt iterations*, are used to achieve time accuracy and are performed on all grids in sequence. This procedure eliminates synchronization errors at blocked and overset boundaries by iteratively bringing all dependent variables up to the next time level.

Figure 6.3 is a diagram of the flow solution process. The inner loop represents the inner iterations or iterations of the SGS solution of the linear equations from one step of the Newton's method. The outer loop represents the dt iterations or steps of the Newton's method.

For time accurate flow calculations with Beggar, it is normal to run more than

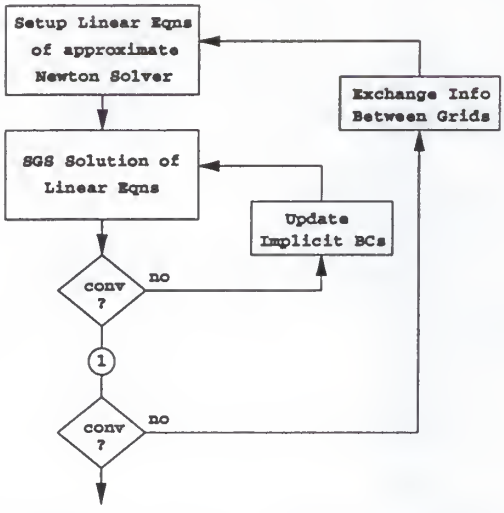


Figure 6.3: Basic flow solution algorithm

one dt iteration to eliminate synchronization errors between grids and to achieve time accuracy. Each dt iteration produces an updated approximation to the flow solution at the next time step. Forces and moments can be calculated after each dt iteration using this approximate solution. If the forces and moments calculated after the first dt iteration are a good approximation to the final forces and moments, these values can be forwarded to the grid assembly process. This allows the computation of the grid assembly to proceed during the computation of additional dt iterations. If the computation time for the flow solver is sufficiently large, the computational cost of the grid assembly process can be hidden.

This implementation is referred to as phase II and is depicted in figure 6.4. Rather than calculating forces and moments only after a complete time step of the flow solver, they are calculated after each dt iteration. The node labeled 1 in figure 6.3 represents the point where the forces and moments are calculated.

Referring to figure 6.4, the solution process is started at time T1. After the first dt iteration, initial approximations to the forces and moments are calculated and are passed from the BE to the FE at time T2. The 6DOF and grid assembly functions proceed while the remaining dt iterations of the flow solver are completed. Once the

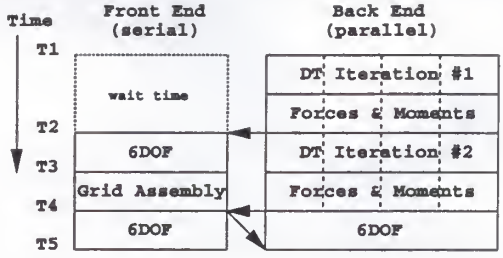


Figure 6.4: Phase II implementation

final forces and moments are calculated, they are passed to the FE process and the 6DOF is repeated. This allows the grids to be moved using the most accurate forces and moments, although the interpolation stencils are updated using grid positions calculated from approximate forces and moments.

The fraction of time spent in computing the flow solution after the first dt iteration is

$$F_t = \left(\frac{F_p}{nbes}\right) * F_i * \left(\frac{ndt-1}{ndt}\right) \tag{6.3}$$

where ndt is the number of dt iterations being run per time step. If F_t is greater than G_s, the time to do the grid assembly can be completely hidden by the time to compute the flow solution, and the speedup is based only on the time to compute the flow solution in parallel. If the time to compute the grid assembly is only partially hidden by the time to compute the flow solution, the speedup is degraded by the portion of the grid assembly process that is not hidden. Thus, the speedup can be approximated by the equation

$$s \simeq \frac{1}{\left(\frac{F_p}{nbes}\right) * F_i + G_t} \tag{6.4}$$

where,

$$G_t = \begin{cases} G_s - F_t & \text{if } F_t < G_s \\ 0 & \text{otherwise} \end{cases} \tag{6.5}$$

Thus, G_t represents the fraction of the grid assembly time that is not hidden by the flow solution. If the grid assembly time is completely hidden, this is the best possible

situation. No matter what technique is used to decompose the work of grid assembly, nothing will do better than reducing the effective execution time to zero. However, as more and more processors are used to reduce the execution time of the flow solver, the time available to hide the execution time of grid assembly decreases.

In figure 6.4, the grid assembly function is finished before the flow solution; therefore, the flow solution proceeds without any delays. However, in figure 6.5, the grid assembly function does not finish before the flow solution. This creates delays in the flow solution function as it waits on information to be communicated from the grid assembly function.

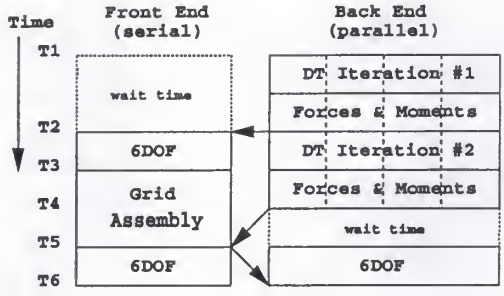


Figure 6.5: Insufficient time to hide grid assembly

By hiding the cost of the grid assembly function, the execution time should be almost equal to that of an equivalent static case in which no grids are moving. In fact, if $G_t = 0$ in equation 6.4, then the speedup is based only on the parallel fraction of the work and a superlinear speedup can be expected, although a decrease in efficiency would be seen, since an additional processor is needed to run the grid assembly function.

Figure 6.6 shows a comparison of the estimated speedup versus processor count for the phase I and phase II implementations as defined by equations 6.1 and 6.4. The curves correspond to work fractions of $F_p = .95$ and $G_s = .05$, a load imbalance factor of $F_i = 1.05$, and $ndt = 2$. The additional processor needed for the grid assembly function in the phase II implementation is better used to do part of the flow solution as long as less than 5 processors are available. Above this point, the phase II

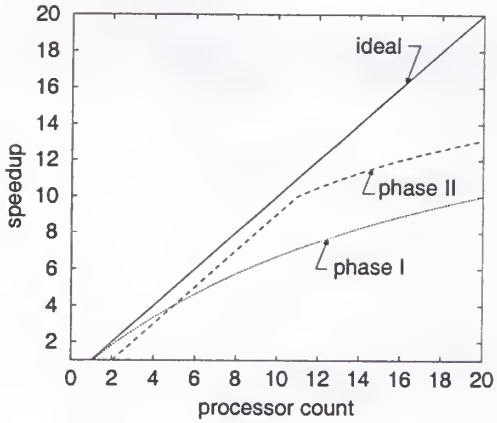


Figure 6.6: Comparison of estimated speedup of phases I and II

implementation outperforms the phase I implementation. The change in slope of the phase II curve (at around 11 processors) is the point where the grid assembly time fails to be hidden by the flow solution time. Above this point, there is a significant dropoff in performance.

Phase III: Coarse Grain Decomposition

As long as the grid assembly time is completely hidden, the optimum speedup is achieved. However, as the number of processors increases, the time to compute the flow solution decreases, the execution time of the grid assembly process is not completely hidden, and the overall performance suffers.

In order to continue to see the optimum speedup, multiple processors must be used to decrease the total execution time of the grid assembly process. This requires consideration of how the grid assembly work and data structures can be distributed across multiple processors.

The work associated with the flow solution is well associated with the grids; therefore, the grids form a good basis for data decomposition of the flow solution. However, the work of the grid assembly function is associated with the number of hole cutting surface facets, the number of cells that are cut, and the number of IGBPs.

This work is not evenly distributed among the grids, and the distribution of work varies as the grids are moved.

For parallel implementation, the primary data structure to be concerned with is the PM tree. This data structure is used during hole cutting and during the search for interpolation stencils. All of the boundaries of the grids are represented by BSP trees stored at the leaf nodes of a single octree. A point is classified against all of the superblocks in the PM tree with a single descent of the octree. This classification identifies if the point is IN or OUT of each superblock. If the point is IN a superblock, then a starting point for stencil jumping is identified (including the correct grid within the superblock).

Since the PM tree is used to classify points relative to superblocks, superblocks were chosen as the basis for coarse grain data decomposition. The superblocks are distributed across multiple FE processes in an effort to equally balance the grid assembly work load. The work load is divided among the processes by cutting holes only into the superblocks mapped to a given process, and only the stencils either donating to or interpolating from these superblocks are identified.

A single octree is used to reduce storage requirements when the complete PM tree is stored with a single process. However, the PM tree is still a major consumer of memory for the grid assembly process. The excess memory requirements must be weighed against the advantages offered by duplicating the entire PM tree on each of the FE processes. If the complete PM tree is available, all possible interpolation stencils for every IGBP within the superblocks mapped to a process can be identified without any communication between processes. Figure 6.7 represents this situation. In this figure, four superblocks are mapped to four FE processes. The superblock represented by the solid line is mapped to the corresponding process. The superblocks represented by the dotted lines are mapped to another process. However, since the complete PM tree is duplicated on each FE process, each process has knowledge of the space occupied by each superblock and can identify all of the interpolation sources

available for the IGBPs in any of the superblocks. Thus, all of the interpolation stencils (represented by the arrows) that donate to IGBPs in superblocks on each processor are identified and the best interpolation source is chosen without any communication. The only communication requirement is the global distribution of the cell state information so that hole points and IGBPs can be identified.

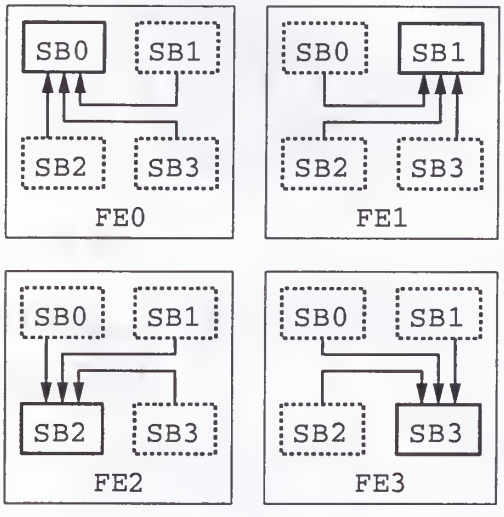


Figure 6.7: Duplication of PM tree on each FE process

Logically, each superblock is represented by a separate PM tree. Therefore, a separate PM tree could be constructed for each superblock and only the PM trees for the superblocks mapped to a process would be stored with that process. This would reduce the memory required per process, but would also increase the amount of communication required. Figure 6.8 represents this situation. With a limited piece of the complete PM tree, a process can only classify points against its superblocks. Thus, only the interpolation stencils from its superblocks, which donate to IGBPs in other superblocks, can be identified. All of the possible interpolation stencils must then be communicated to the process which owns the superblocks that will receive the donations, so that the best source can be identified. The increase in communications and the coding changes required to implement separate PM trees have driven the decision to duplicate the complete PM tree on each of the FE processes.

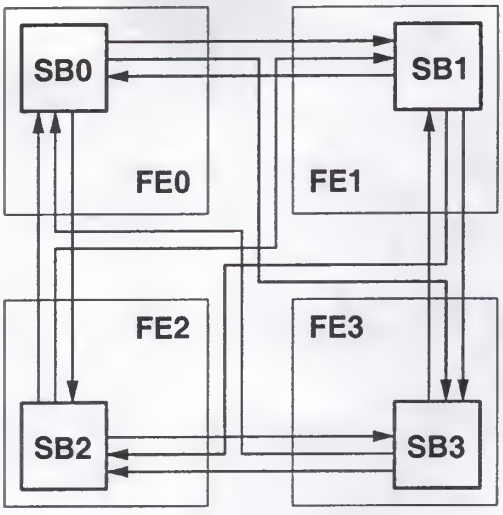


Figure 6.8: Distribution of PM tree across the FE processes

The phase III implementation is shown in figure 6.9. This is essentially the same as figure 6.4 except that the grid assembly function is distributed across multiple processors. A load balancing function has also been added. Since the work load is dynamic, the grid assembly function is monitored to judge the load balance, and dynamic load balancing is performed by moving superblocks between processors. The monitoring required to judge the load balance is accomplished by measuring the execution time of the grid assembly using system calls. These execution times are measured on a superblock by superblock basis and are passed into the dynamic load balancing function described in chapter 5. The load balancing function is executed during the calculation of the initial dt iteration; therefore, if the load balancing function is cheap, its execution time will be hidden and will not adversely impact code performance.

Since the grid assembly is now executed in parallel, the grid assembly execution time is represented by

$$\frac{G_p}{nfes} * G_i \tag{6.6}$$

where G_p is the parallel fraction of the work represented by the grid assembly func-

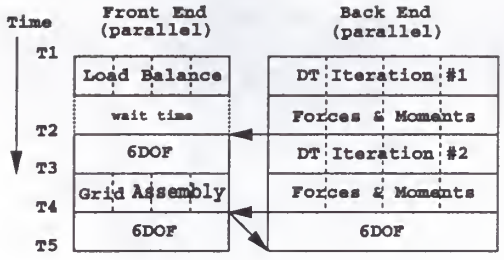


Figure 6.9: Phase III implementation

tion, G_i is the load imbalance in distributing the grid assembly work, and $nfes$ is the number of FE processors that are executing the grid assembly function. Functional overlapping of the flow solution and grid assembly functions is still being used; therefore, the speedup is still estimated using equation 6.4 with

$$G_t = \begin{cases} \frac{G_p}{nfes} * G_i - F_t & \text{if } F_t < \frac{G_p}{nfes} * G_i \\ 0 & \text{otherwise} \end{cases} \tag{6.7}$$

Figure 6.10 compares the estimated speedup of the phase III implementation to that of phases I and II. The curve plotted for phase III is for $G_p = 0.05$, $nfes = 4$, and $G_i = 1.$ (ideal load balance). The curves plotted for phases I and II are the same as those plotted in figure 6.6. Since the grid assembly is executed on 4 processors, at least 5 processors are required to execute the phase III implementation. About 12 processors are needed before phase III outperforms phase I and about 16 processors are needed before phase III outperforms phase II. More than 40 processors are required before the grid assembly time fails to be hidden by the flow solution time; however, if the grid assembly function stays well balanced, more FE processes could be added to produce solutions that scale to higher processor counts.

Phase IV: Fine Grain Decomposition

The relatively small number of superblocks, used for most configurations, limits the ability to achieve a good load balance of the grid assembly work load. The splitting of grids, which is used to help balance the load of the flow solver, does not introduce

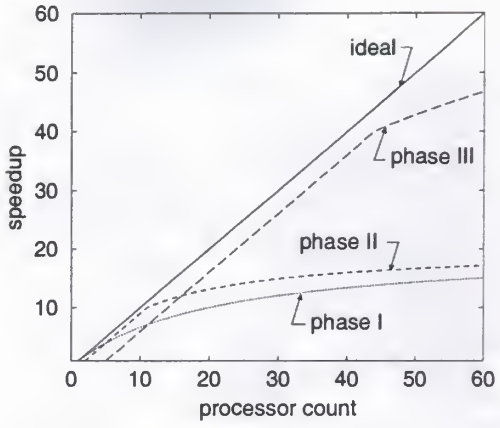


Figure 6.10: Comparison of the estimated speedup of phases I, II, and III

new superblocks and therefore, is of no help in load balancing the grid assembly work load. Instead, a finer grain decomposition of the work of the grid assembly function must be used to ensure the possibility of load balancing the grid assembly work load on any large number of processors.

The two most expensive components of the grid assembly function are hole cutting and the search for interpolation stencils. The work of the hole cutting function is associated with the number of hole cutting facets. The work of the search for interpolation stencils is associated with the number of IGBPs that require interpolation. Therefore, a fine grain decomposition of the grid assembly function may be based on the view that the smallest piece of work is a single hole cutting facet or an IGBP. Load balancing is achieved by equally distributing the total number of hole cutting facets and IGBPs across the available FE processes. Each facet and each IGBP is independent of its neighbors; therefore, there is no communication between neighboring facets or IGBPs. The only area of concern is the access and updating of several data structures by multiple processes.

The hole cutting facets are defined by the solid surface boundary conditions as specified by user input. These boundary conditions are specified as grid surface segments, and the specifications are stored as a linked list. The facets are not stored

as a single array, but the total number of facets can be counted, and as the linked list of specifications is traversed, each process will cut holes into all of the overlapping grids using its share of the facets.

As described in chapter 2, each hole cutting facet marks a part of the hole outline, independent of the other facets. The hole cutting algorithm identifies a set of cells near the facet and compares the cell centers to the plane definition of the facet. Some cells will be marked as holes, others will be marked as worldside. This information is called *cell state* information and is stored in an array associated with each grid. These two pieces of information are stored separately (as bit flags) so that a cell can be marked as a hole and as a worldside point (i.e. a worldside hole). This is because one facet may mark a cell as a hole, and another facet may mark it as worldside. The worldside status is needed to cap off the holes during the hole filling process, while the hole status is needed to get the hole filling process started.

When hole cutting is done with two neighboring facets on separate processors, the two cell state arrays must be merged to define the complete hole outline before the hole filling can be done. This merge is accomplished by a bitwise OR of the cell state status bits. This is a reduction operation and involves combining separate cell state arrays from each process for every grid in the system. This can be a rather expensive operation. For this reason, the cell state arrays will be stored in shared memory.

When using shared memory, some facility is often required to make sure that only one process changes a variable at a time. However, in the case of hole cutting, it doesn't matter which facet marks a cell as a hole or as worldside. No matter how many facets mark a cell as a hole or worldside, the cell state information is only a set of bit flags. It doesn't matter how many processes set a bit as long as it gets set (no one ever clears these flags during grid assembly). Likewise, the order in which processes set a cell's status bit is immaterial. Thus, the cell state information is stored in shared memory and no coordination between processes is needed to ensure that

the cell state information is constructed correctly.

The use of IGBPs for fine grain decomposition of the stencil jumping requires access to the complete PM tree. Without access to the complete PM tree, each process will be limited as to which IGBPs it can process. However, the duplication of the PM tree across multiple processes would require a large amount of memory, as stated before. Therefore, the storage of the PM tree in shared memory is needed for the fine grain decomposition of the stencil search based on IGBPs. Once the PM tree is built, it does not require any modifications. Each process can access the PM tree without any need for communication, cooperation, or synchronization.

The use of a fine grain decomposition of the work associated with grid assembly should allow better load balancing on larger processor counts. Therefore, overlapping of the grid assembly time and the flow solution time could be used, with the increased number of processors used to decrease the grid assembly time, so that it continues to be hidden. However, if the fine grain decomposition allows for a good load balance without excess overhead, the execution model could revert back to each function being spread across all of the available processors as shown in figure 6.11. This would allow complete time accurate updating of the grid assembly.

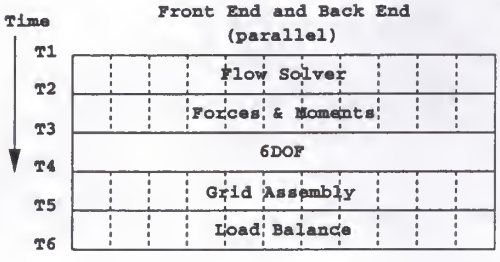


Figure 6.11: Phase IV implementation

Figure 6.12 shows a comparison of the estimated speedup of phases I-IV. The curves for phases I-III are the same as those shown in figure 6.10 with $F_p = 0.95, F_i = 1.05, G_s = 0.05$ for phases I and II, and $G_p = 0.05, G_i = 1., nfes = 4$, and $ndt = 2$

for phase III. The phase IV curve is defined using the equation

$$s \simeq \frac{1}{\left(\frac{F_p}{npes}\right) * F_i + \left(\frac{G_p}{npes}\right) * G_i} \tag{6.8}$$

where $F_p = 0.95, F_i = 1.05, G_p = 0.05, G_i = 1.20$, and $npes$ is the total number of processors used. Even with a 20% imbalance in the grid assembly and the lack of execution time overlapping, this implementation would appear to outperform the others.

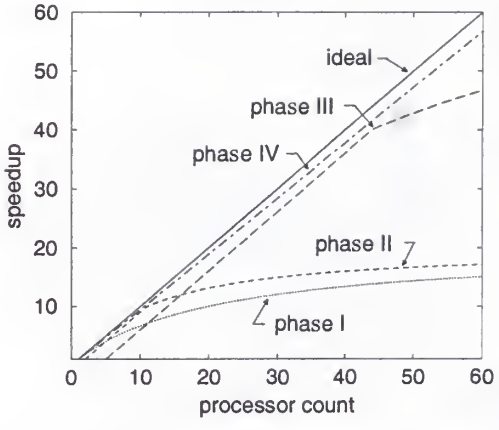


Figure 6.12: Comparison of estimated speedup of phases I, II, III and IV

However, note that as the processor count increases, the slope of the phase IV curve causes it to diverge from the ideal speedup curve. On the other hand, the phase III curve runs parallel to the ideal speedup curve up to the point at which the grid assembly time fails to be hidden by the flow solution time. This behavior is due to the overlapping of the grid assembly time by the flow solution time rather than any overhead introduced in the implementation of phase III. Thus, if overhead due to the fine grain decomposition of phase IV and the distribution of all of the functions across all of the processors becomes large, the overlapping of the grid assembly time with the flow solution time may offer some improvements in scalability.

Summary

Table 6.1 gives a summary of the four different implementations. The first implementation was used to begin solving dynamic problems on a parallel computer and to begin the analysis of the parallel processing performance. The second implementation improved performance by hiding some of the execution time of the grid assembly function behind the execution time of the flow solver. The third implementation attempts to decrease the total execution time of the grid assembly function by using multiple FE processes and a coarse grain decomposition of the grid assembly work based on superblocks. The final implementation uses a fine grain decomposition of the work of grid assembly based on hole cutting facets and IGBPs. The goal is to use all of the available processors for both the grid assembly and the flow solution.

Table 6.1: Summary of the implementations of parallel grid assembly

Phase	Description
I	Single process performs complete grid assembly serially with respect to the parallel execution of the flow solver.
II	Single process performs complete grid assembly based on approximate forces and moments. Overlapping of grid assembly time with flow solution time is used.
III	Multiple processes perform grid assembly in parallel using coarse grain decomposition based on superblocks. Overlapping of grid assembly time and flow solution time is continued.
IV	Multiple processes perform grid assembly in parallel using fine grain decomposition. Load balancing of hole cutting is separate from that of the stencil search.

CHAPTER 7
TEST PROBLEM

The three store, ripple release case, first presented in Thoms and Jordan [30] and later modelled using Beggar in Prewitt et al. [4], is being used as a test case for timing each of the parallel implementations. The geometric configuration is that of three generic stores in a triple ejector rack (TER) configuration under a generic pylon attached to a clipped delta wing. This configuration is depicted in figures 1.1 and 1.3 with the stores under the right wing. The three generic stores are identical bodies of revolution with an ogive-cylinder-ogive planform shape and four clipped delta fins with NACA 0008 airfoil cross section. The pylon is an extruded surface of similar ogive-cylinder-ogive cross section. The wing has a 45 degree leading edge sweep and a NACA 64A010 airfoil cross section.

The mass properties of the three stores are listed in table 7.1. The products of inertia that are not shown are zero due to symmetry. The CG is located on the axis of revolution, 4.65 ft aft of the nose. The reference length is equal to the store diameter of 20 inches and the reference area is equal to the cross sectional area at the axial location of the CG.

Table 7.1: Store physical properties

Property	Value
weight	1000 lb
I_{xx}	10 slug $\cdot$ ft^2
I_{yy}	180 slug $\cdot$ ft^2
I_{zz}	180 slug $\cdot$ ft^2
CG location	4.65 ft aft of the nose
ref. length	1.6667 ft
ref. area	2.1817 ft^2

Ejectors are used to help ensure a safe separation trajectory. The properties of the ejectors are listed in table 7.2. Each store is acted on by a pair of ejectors that create a nose up pitching moment to counteract a strong aerodynamic nose down pitching moment seen in carriage. The ejectors are directed downward on the bottom store of the TER and are directed outward at 45 degrees (with respect to vertical) on the two shoulder stores. The stores are released in bottom-outboard-inboard order with a 0.04 sec delay between each release. The ejectors are applied at release for a duration of 0.045 sec. In the simulations, the ejectors are defined fixed relative to the stores so that the ejectors will not create a rolling moment due to store motion.

Table 7.2: Ejector properties

		Bottom	Outboard	Inboard
Release time		0 sec	0.04 sec	0.08 sec
Forward ejector	force	1800 lb	1800 lb	1800 lb
	location	4.06 ft aft of the nose		
	direction	$+\hat{e}_z$	$+\frac{1}{\sqrt{2}}\hat{e}_y + \frac{1}{\sqrt{2}}\hat{e}_z$	$-\frac{1}{\sqrt{2}}\hat{e}_y + \frac{1}{\sqrt{2}}\hat{e}_z$
	duration	0.045 sec	0.045 sec	0.045 sec
Aft ejector	force	7200 lb	7200 lb	7200 lb
	location	5.73 ft aft of the nose		
	direction	$+\hat{e}_z$	$+\frac{1}{\sqrt{2}}\hat{e}_y + \frac{1}{\sqrt{2}}\hat{e}_z$	$-\frac{1}{\sqrt{2}}\hat{e}_y + \frac{1}{\sqrt{2}}\hat{e}_z$
	duration	0.045 sec	0.045 sec	0.045 sec

The original grids for this configuration used separate blocked grid systems around the stores, pylon, and wing. Each of these blocked grid systems forms a separate superblock. Additional interface grids were used to improve the flow solution and to increase the grid overlap required to ensure successful assembly of the grid system. Each store grid consisted of four blocks, one between each pair of fins, defining one quarter of the geometry. The wing and wing/pylon interface grids were generated as single grids. Due to Beggar's use of component grids for coarse grain decomposition of the flow solver, the large wing and wing/pylon interface grids were split into three blocks each. Since Beggar allows block-to-block boundary connections, splitting a grid into smaller blocks introduces new boundary conditions, but

does not adversely affect the ability for a grid system to assembly. Table 7.3 lists the block sizes of the original grids. The largest grid in this grid system has 179,550 grid points. With the load balance determined based solely on the number of grid points per process, this grid system will load balance well for 12-14 processes. Some of the initial timing runs were done using this set of grids.

Table 7.3: Original grid dimensions

Superblock	Block dimensions	Total Points
bottom store	4 @ 116x38x21	4x92,568
outboard store	4 @ 116x38x21	4x92,568
inboard store	4 @ 116x38x21	4x92,568
pylon	149x33x15	73,755
	61x10x15	9,150
wing	135x14x95	179,550
	135x10x95	128,250
	135x9x95	115,425
wing/pylon interface	2 @ 111x28x51	2x158,508
	111x27x51	152,847
fin tip interface	49x24x22	25,872
outer bndry interface	58x29x10	16,820
fin tip interface	49x24x22	25,872
outer bndry interface	58x29x10	16,820
10 superblocks	24 blocks	2,172,193

In order to effectively utilize larger numbers of processors, smaller grids are needed to load balance the flow solver. Table 7.4 lists a new set of grids generated by splitting the existing grids. This introduces new block-to-block boundaries; thus, the number of superblocks stayed the same, but the number of blocks increased from 24 to 67. This also increases the total number of grid points, although the number of grid cells has not increased. The largest grid in this grid system has 62,370 grid points. Therefore, this set of grids should extend the processor count beyond 32.

No particularly intelligent scheme was used to split up the grids. If a grid is split along its largest dimension, the block-to-block boundary introduced will represent the smallest possible surface area; thus, the cost of implementing this boundary condition will be minimized. Conversely, some splittings may have less effects on the flow

solution convergence than others. However, in this case, the splitting of the grids was done by hand, and the grids were often split so as to minimize the amount of work required to change the input files. Since block-to-block boundary conditions are detected automatically, but solid surface boundary conditions have to be specified, the splitting was often done to reduce the splitting of solid surface boundaries. This may also have some beneficial effects on the flow solver since the implicit treatment of the solid surface boundary conditions can be done in larger pieces.

Table 7.4: Dimensions of split grids

Superblock	Block dimensions	Total Points
bottom store	4 @ 71x38x21	4x56,658
	4 @ 26x38x21	4x20,748
	4 @ 21x38x21	4x16,758
outboard store	4 @ 71x38x21	4x56,658
	4 @ 26x38x21	4x20,748
	4 @ 21x38x21	4x16,758
inboard store	4 @ 71x38x21	4x56,658
	4 @ 26x38x21	4x20,748
	4 @ 21x38x21	4x16,758
pylon	2 @ 75x33x15	2x37,125
	61x10x15	9,150
wing	4 @ 135x14x32	4x60,480
	2 @ 135x14x33	2x62,370
	5 @ 135x9x17	5x20,655
	135x9x15	18,225
wing/pylon interface	4 @ 111x28x18	4x55,944
	2 @ 111x28x17	2x52,836
	5 @ 19x27x51	5x26,163
	21x27x51	28,917
fin tip interface	49x24x22	25,872
outer bndry interface	58x29x10	16,820
fin tip interface	49x24x22	25,872
outer bndry interface	58x29x10	16,820
10 superblocks	67 blocks	2,276,092

Since the work of the flow solver is closely associated with the number of grid points, the load balance of the flow solver work can be judged by the distribution of the grid points. Table 7.5 lists the load imbalance factors achieved based solely on

numbers of grid points per processor and using the 67 block grid system. The third column lists the effective number of processors, which is equal to the actual number of processors divided by the load imbalance factor. As can be seen, the flow solver should load balance relatively well up to 40 processors. Beyond this point, an increase in the processor count is offset by the load imbalance.

Table 7.5: Load Imbalance Factors

no. of processors	F_i	effec. no. of processors
4	1.005	3.98
8	1.04	7.69
12	1.05	11.43
16	1.05	15.24
20	1.07	18.69
24	1.06	22.64
28	1.14	24.56
32	1.07	29.91
36	1.18	30.51
40	1.12	35.71
44	1.23	35.77
48	1.35	35.56

The flight conditions simulated in all of the test runs are for a freestream Mach number of 0.95 at 26,000 feet altitude. All solutions are calculated assuming inviscid flow. The flow solver is run time-accurately with a time step of 0.0005 sec, and a local time step, based on a CFL number of 2, is used to accelerate convergence of the dt iterations. Two dt iterations are used per time step and six inner iterations per dt iteration. A total of 600 iterations are run giving a total of 0.3 seconds of the trajectory. The flow solver is run with second order spatial accuracy using Steger-Warming flux jacobians, primitive variable MUSCL extrapolation, flux difference splitting with Roe averaged variables, and the van Albada flux limiter. Implicit, solid-wall boundary conditions are used with a relaxation factor of 0.6. All solutions are run in double precision (64 bit) on an SGI Origin 2000 machine. This particular machine is configured with 64 - 195 MHz R10000 processors and 16 Gb of shared memory distributed

as 2 processors and 512 Mb per node card.

A single processor simulation was computed to establish the base solution time of 9384 minutes (about 6.5 days) using the 24 block grid system. The execution time of the grid assembly function was compared to the total execution time from the sequential run to establish work fractions of $G_s = .05$ and $F_s = .95$ for the grid assembly and the flow solver, respectively.

The single processor solution was repeated with the 67 block grid system and the execution time increased to 9434 minutes. This increase is due to a change in the grid system that is only needed to improve performance for parallel execution. Therefore, all timings will be compared to the faster sequential time of 9384 minutes. Table 7.6 gives a summary of the final position of the stores after 0.3 seconds of the trajectory as computed with the 24 block grid and the 67 block grid. This illustrates the order of magnitude of the changes that can be expected in the final solution by introducing block-to-block boundaries by splitting grids. The largest changes are on the order of tenths of a degree, while most changes are much smaller.

Table 7.6: Summary of the final position of the stores calculated from the two different grid sets

		24 blocks			67 blocks		
		bottom	outboard	inboard	bottom	outboard	inboard
position	x	-1.5867	-1.2077	-0.9800	-1.5868	-1.2078	-0.9935
	y	-0.0007	2.9046	-2.9831	-0.0010	2.9053	-2.9851
	z	6.5293	2.9240	2.0040	6.5311	2.9227	2.0075
angles	yaw	8.0665	2.4242	16.6171	8.0537	2.4752	16.4353
	pitch	3.4369	0.3671	-18.5046	3.4349	0.1890	-18.6291
	roll	1.3669	-0.4678	-4.6363	1.3667	-0.5366	-4.6459

Figures 7.1–7.3 present the trajectories that were calculated on a single processor and presented in Prewitt et al. [4]. CG locations and the angular position of the stores during the ripple release trajectory calculation are presented. All three stores move downward and downstream. The bottom store shows only a slight sideways motion, while the outboard store moves further outboard and the inboard store moves further

inboard, both due to the ejector forces. The inboard motion of the inboard store is quickly reversed due to aerodynamic forces. All three stores are pitched nose up by the ejectors before pitching nose down, although the inboard store only pitches slightly nose up before quickly approaching 25 degrees nose down. The bottom and inboard stores yaw nose outboard and roll lugs outboard. The outboard store rolls lugs inboard and yaws nose inboard before turning nose outboard.

The accuracy of the solutions from the parallel runs will be summarized, but corresponding plots of the trajectory data will not be shown. This is because the solutions from the parallel runs are nearly identical to the solutions from the sequential run, and the curves are not distinguishable on plots of this scale. The solutions can be expected to change slightly because the grids are distributed differently, based on the number of processors being used. This affects the explicit passing of information between grids on different processors and thus can affect the flow solution. However, the effects seen on the final trajectory are minimal.

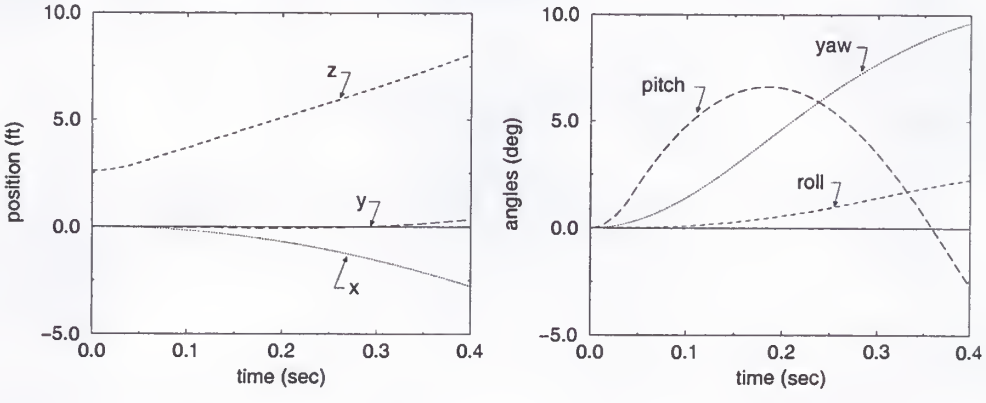


Figure 7.1: Bottom store (left) CG and (right) angular positions

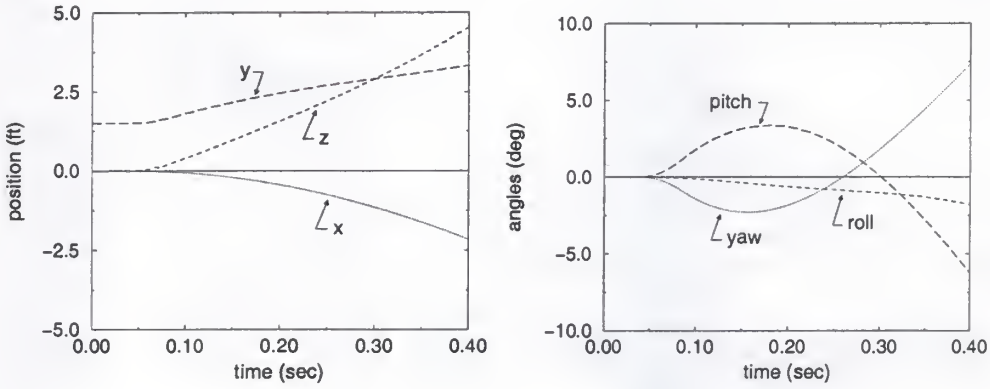


Figure 7.2: Outboard store (left) CG and (right) angular positions

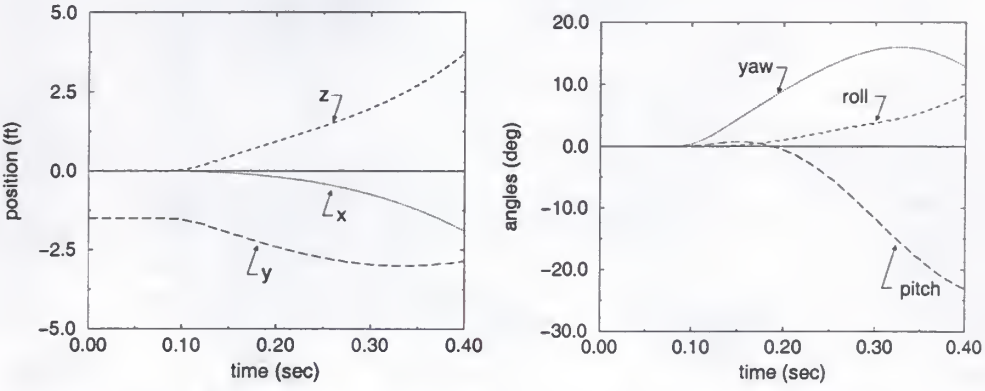


Figure 7.3: Inboard store (left) CG and (right) angular positions

CHAPTER 8
RESULTS

Phase I: Hybrid Parallel-Sequential

The ripple release test problem was run on the SGI Origin 2000 using the phase I implementation and the 24 block grid system. The grid assembly was run in a single FE process, and the flow solver was decomposed and executed using 4, 8, 12, and 16 BE processes. The timing results are displayed in figure 8.1. The actual speedup is plotted against the estimated speedup as defined by equation 6.1. In this implementation, the grid assembly for the entire domain is performed by a single process, that executes sequentially with respect to the parallel execution of the flow solver. Once a time step of the flow solver is complete, the grid assembly process is swapped in to perform the grid assembly. The grid assembly function and flow solver do not execute at the same time, thus the speedup data are plotted against the number of BE processes, which is equivalent to the maximum number of processes that are running at any one time.

For the estimated speedup curve, the load imbalance factor for the flow solver ($F_i = 1.05$) represents a nominal imbalance of 5%. The speedup for these cases is actually better than the predicted value. This is most likely due to a latency experienced on the SGI Origin 2000 architecture. When a processor accesses memory off of its node card, there is a delay when compared to accessing the memory on its node card. As more processors are used, the amount of memory, that a processor needs, decreases since the grids are distributed across the available processors. Thus, the potential that the data can be stored in the memory on the node card increases. The large memory job running on a single processor runs slower due to this latency;

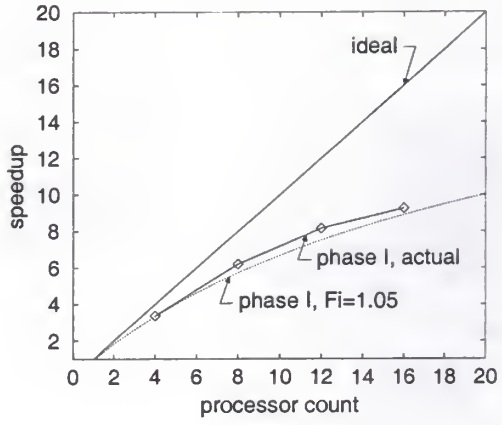


Figure 8.1: Actual speedup of phase I

thus, an artificial speedup is seen with increased processor counts. The estimated speedup curve still follows the trends in the data very well.

The accuracy of the solution is demonstrated by the position data shown in table 8.1. These data represent the position of the CG and angular orientation of the bottom store after 0.3 seconds of the trajectory. The maximum difference, as compared to the sequential run, is on the order of 0.002 feet and 0.03 degrees. The actual execution times (in minutes) are also listed in table 8.1.

Table 8.1: Summary of results from the phase I runs including the final position of the bottom store

		Seq	Phase I			
no. of processors		1	4	8	12	16
position	x	-1.5867	-1.5867	-1.5867	-1.5868	-1.5871
	y	-0.0007	-0.0005	-0.0005	-0.0004	-0.0012
	z	6.5293	6.5292	6.5292	6.5292	6.5311
orientation	yaw	8.0665	8.0659	8.0659	8.0677	8.0456
	pitch	3.4369	3.4347	3.4349	3.4355	3.4295
	roll	1.3669	1.3336	1.3343	1.3587	1.3690
exec. time (min)		9384	2786	1513	1150	1013

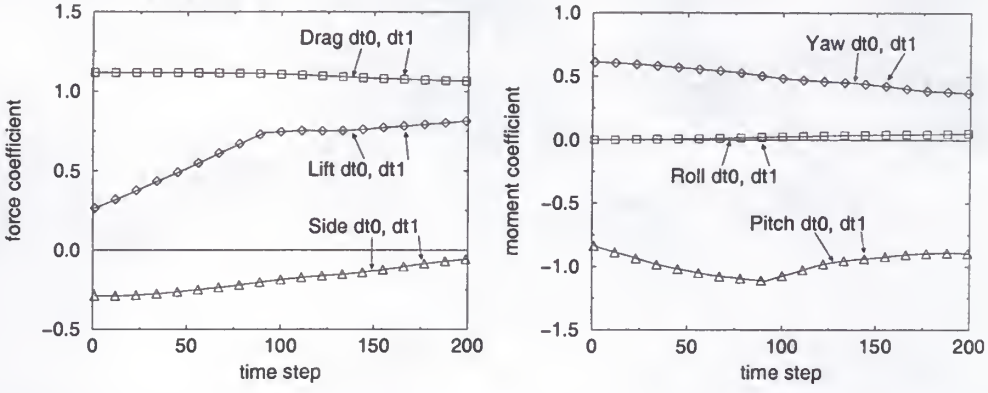


Figure 8.2: Bottom store (left) force coefficient and (right) moment coefficient variation between dt iterations history

Phase II: Function Overlapping

This implementation continues to use a single process to compute the grid assembly for the entire domain. However, some parallel performance is gained by overlapping the execution of the grid assembly and the flow solver. The approach of overlapping the grid assembly and flow solution functions is worthless if the forces and moments, calculated after the first dt iteration, are not a good approximation of the final forces and moments to be calculated. Thus, several test cases were run to monitor the forces and moments after each dt iteration. Figures 8.2–8.4 show a time history of the force and moment coefficients for the three stores after each dt iteration of 200 time steps of the separation trajectory. All of the force and moment coefficients show good agreement between dt iterations. As an example, the maximum variation (between dt iterations) in the pitching moment coefficient for the bottom store throughout the entire trajectory calculation (600 time steps) was only 0.2%.

It might be deduced that the use of implicit solid wall boundary conditions helps to accelerate the convergence of the flow solution near the walls, although additional dt iterations are required to ensure convergence throughout the domain. Therefore, the most likely parameters to affect the forces and moments are the number of inner iterations and the BC relaxation factor; thus, tests were repeated with variations

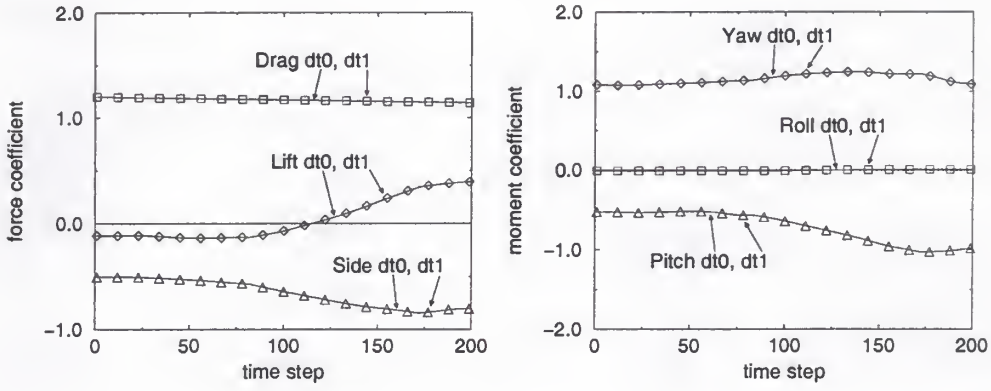


Figure 8.3: Outboard store (left) force coefficient and (right) moment coefficient variation between dt iterations history

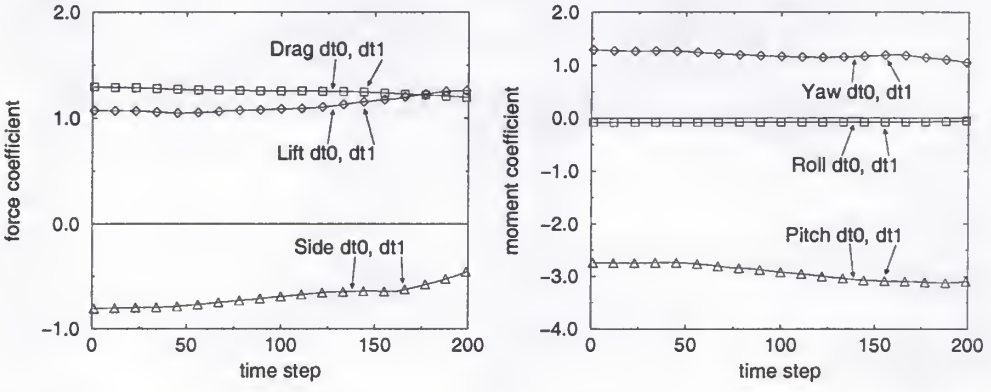


Figure 8.4: Inboard store (left) force coefficient and (right) moment coefficient variation between dt iterations history

in the number of inner iterations from 3 to 6 and in the BC relaxation factor from .6 to 1. For all test cases, the forces and moments behaved similarly. In fact, the forces and moments are so well behaved for this test problem, it is conceivable that an extrapolation procedure could be used to give a better approximation to the final forces and moments, using the initial forces and moments and some history from previous time steps.

The timing results from the phase II runs are presented in figure 8.5. This shows the actual speedup versus processor count as compared to the estimated speedup defined by equations 6.4 and 6.5. The ripple release test case was run using 4, 8, 12, and 16 BE processes for the flow solver and a single FE process for grid assembly. Since the grid assembly function is executing at the same time that the flow solver is executing, the total number of processors running at any time must now include the grid assembly process. Therefore, the actual speedup data points are plotted at 5, 9, 13, and 17 processors. Again, an artificial speedup is seen in the results, but the estimated speedup curve follows the trends of the data. For the phase II runs on 4+1 (4 BE processes and 1 FE process) and 8+1 processors, the grid assembly time was completely hidden by the execution time of the flow solver. On 12+1 and 16+1 processors, the grid assembly time was not completely hidden, and the performance suffers.

One troublesome result is the speedup of the 12+1 processor run. Why did this result fall below the estimated speedup curve? This calculation was repeated several times and the results were consistent. This is an artifact of using average execution times, over the complete simulation, to determine the work fractions of grid assembly and the flow solver. As will be seen later, the execution time of the grid assembly actually decreases throughout the simulation. When the stores are in carriage position, the grid assembly work is at a maximum. As the stores move downward, some of the grids no longer overlap, less holes are cut, less stencil sources are searched, and the execution time decreases. To see how the use of an average

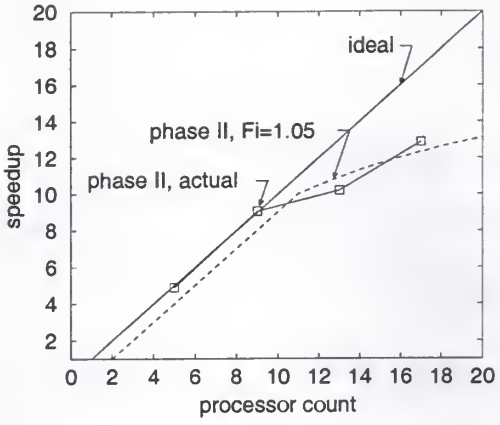


Figure 8.5: Actual speedup of phase II

execution time can affect the estimated speedup, consider figure 8.6.

This figure represents the variation in the grid assembly time by the sloping lines labeled "GA". The average grid assembly time is represented by the dashed lines labeled "Avg GA". The constant flow solution time, available to hide the execution of the grid assembly, is represented by the horizontal lines labeled "Flow". For the two plots in the upper half of the figure, the flow solution time is always greater than the grid assembly time, or it is always less than the grid assembly time. In these cases, the relationship between the flow solution time and the grid assembly time is accurately modelled by the use of average execution times. However, in the lower two plots, the execution time of the flow solver is sufficient to hide the execution time of the grid assembly for part of the time steps, but is insufficient for the rest. In the case represented by the plot in the lower left corner of the figure, if the average execution time of the grid assembly is compared to the flow solution time, it appears that the grid assembly is always hidden by the flow solver. However, for the time steps below what is labeled "n1", the grid assembly execution time is not hidden. Thus the actual speedup would be less than that predicted by equations 6.4 and 6.5. In the case represented by the plot in the lower right corner of the figure, it appears that the correct conclusion would be drawn by considering the average execution time.

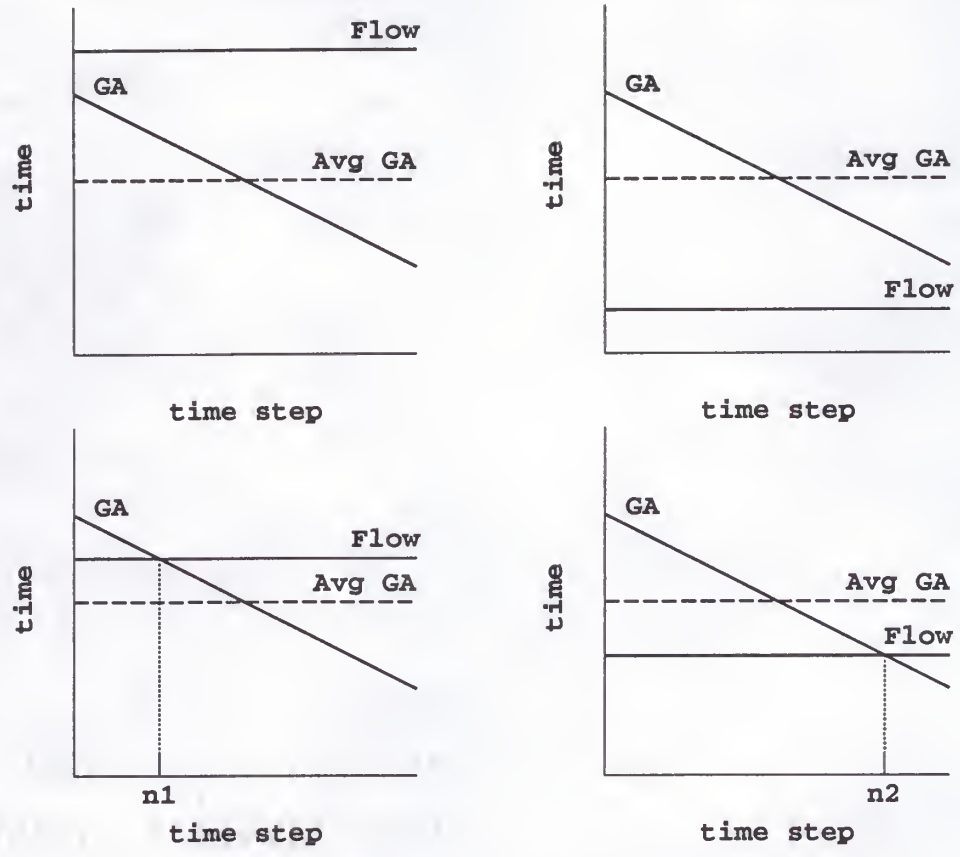


Figure 8.6: Effect of using average execution time

However, the speedup estimated would be high, because of the use of the time steps above what is labeled "n2" when computing the average grid assembly time.

This exercise points out that the equations developed to estimate the speedup are only applicable for constant execution times (or work fractions). If the execution times are not constant, the equations should be applied on an time step by time step basis, where the work fractions are a function of the time step number. However, since the equations are only used to judge if the actual implementation performs as expected, they can still be applied. They may also be of use in a production work environment to estimate code performance on particular problems based on past experience of the work fractions and load imbalance factors.

The accuracy of the solutions from the phase II runs can be judged by the data shown in table 8.2. These data represent the position of the CG and angular orientation of the outboard store after the complete trajectory. The maximum difference, as compared to the sequential run, is on the order of 0.001 feet and 0.15 degrees. The use of functional overlapping to hide the execution time of the grid assembly function has not deteriorated the accuracy of the results for this test problem.

Table 8.2: Summary of results from the phase II runs including the final position of the outboard store

		Seq	Phase II			
no. of processors		1	4+1	8+1	12+1	16+1
position	x	-1.2077	-1.2077	-1.2078	-1.2078	-1.2091
	y	2.9046	2.9049	2.9050	2.9051	2.9050
	z	2.9240	2.9233	2.9235	2.9235	2.9235
orientation	yaw	2.4242	2.4243	2.4252	2.4199	2.4983
	pitch	0.3671	0.3840	0.3875	0.3910	0.2103
	roll	-0.4678	-0.4793	-0.4759	-0.4697	-0.5251
exec. time (min)		9384	1920	1041	920	729

Phase III: Coarse Grain Decomposition

This implementation uses multiple FE processes to reduce the grid assembly time. Functional overlapping is still being used in an attempt to hide the grid assembly time behind the execution time of the flow solver. The decomposition of the grid assembly work is based on superblocks. The relatively small number of superblocks would classify this as a coarse grain decomposition technique. Dynamic load balancing, as described in chapter 5, is used to shuffle the superblocks between processes in order to balance the distribution of work based on some measure of the work per superblock. Since no apriori metric exists for judging the work of grid assembly, the actual execution time, as measured using system calls, is used as the measure of the work associated with a superblock.

The ripple release test problem was run using the phase III implementation and

the 67 block grid system. The timing results are presented in figure 8.7. All of the runs used 4 FE processes and the number of BE processes varied between 16 and 40. Dynamic load balancing of the grid assembly function was performed after each time step. The load balance of the grid assembly was judged by measuring the execution times of the hole cutting, stencil searching, and interpolation health check routines. Figure 8.8 shows a time history of the grid assembly load imbalance factor for the 40+4 processor run. The average load imbalance in the grid assembly appears to be less than 10%. Thus, one of the estimated speedup curves plotted in figure 8.7 is for load imbalance factors of $F_i = 1.05$ and $G_i = 1.08$. The speedup experienced followed the estimated speedup up to 24+4 processes; however, for larger processor counts, the grid assembly time failed to be hidden, and the speedup fell well below the estimated value.

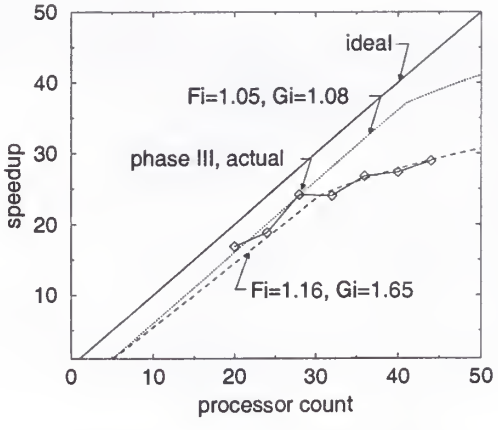


Figure 8.7: Actual speedup of phase III

The imbalance in the grid assembly work load is obviously worse than the measured execution times would imply. It appears that the 24+4 processor run is the last point where the grid assembly time is hidden by the execution time of the flow solver. If we assume that the grid assembly time from the 24+4 processor run exactly equals the execution time of the flow solver that is available to hide the grid assembly, then the imbalance factor can be calculated. Thus, the grid assembly imbalance was

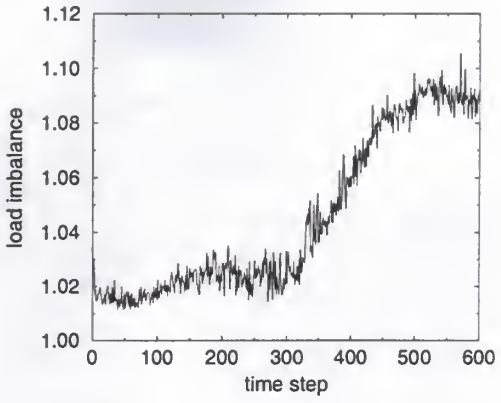


Figure 8.8: History of grid assembly load imbalance based on execution times of hole cutting, stencil search, and health check

calculated using equations 6.4 and 6.7 and the timings of the 24+4 processor run. This showed the actual imbalance in the grid assembly is about 65%.

The second estimated speedup curve in figure 8.7 represents imbalance factors of $F_i = 1.16$ and $G_i = 1.65$. This is based on the imbalance in the grid assembly calculated from the timing results of the 24+4 run and a more realistic value for the imbalance factor for the flow solver on larger processor counts (see table 7.5). This curve matches the four jobs with the larger processor counts more closely.

The cause for the difference in the perceived imbalance, as judged by the measured execution times of the grid assembly functions, and the actual imbalance in the grid assembly function, as calculated from the estimated speedup equations, is the need for synchronization. Without synchronization, the total measured execution time per process may be well balanced, but if synchronization is required between the functions, wait time can be introduced if each function is not well balanced. This point was discussed in chapter 5 and is illustrated in figure 5.3.

Figure 8.9 represents the grid assembly process. After the holes are cut, control enters into an iterative loop in which interpolation stencils for IGBPs are identified. If any IGBP fails to have an interpolation source, it is marked as OUT and neighboring points become IGBPs requiring interpolation. Once a valid set of interpolation sten-

cils are identified, the health of the interpolation stencils is checked to chose the best interpolation source. The most expensive two functions in the process are cutting holes and the search for interpolation stencils. The most expensive of the remaining functions is the check of the interpolation stencil health. Together these three functions account for 90%-95% of the execution time. Therefore, that is why the sum of these execution times was chosen to judge the load balance.

This iterative algorithm requires several synchronization points to ensure that each process has access to the proper cell state information and to ensure that each process executes the same number of iterations of the loop. The processes are synchronized after the holes are cut, so that each process will know which grid points are IGBPs and which cells can not be interpolated from. Likewise, each iteration of the loop has to be synchronized whenever an IGBP is marked as OUT because it either failed to have any interpolation sources or the interpolation stencils did not meet the health requirements. These synchronization points mean that the total execution time should not be used to judge the load balance.

Since the stencil search function was the most expensive function for the ripple release test problem, the execution time of just the stencil search function was used in subsequent runs to judge the grid assembly load balance. Figure 8.10 shows a plot of the perceived load imbalance. Again, grid assembly appears to be relatively well balanced, and the overall execution times were nearly equivalent to those plotted in figure 8.7.

Figure 8.11 shows a time history of the grid assembly times measured on the 4 FE processes of the 40+4 processor run. The data are from a run in which the load balance was judged by the execution time of the stencil search routine. Each plot in the figure represents a different process. The curves labeled "total" represent the total execution time of the grid assembly function, which decreases as the calculation progresses, since the stores begin to move downward and some of the grids no longer overlap. The curves labeled "flow" represent the time to calculate one dt iteration

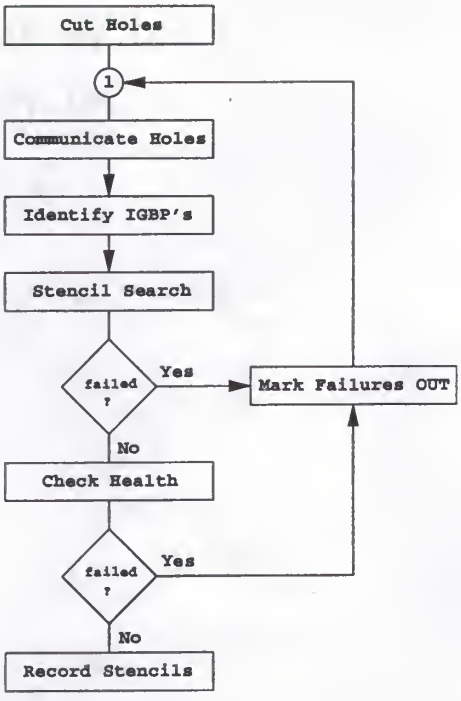


Figure 8.9: Grid assembly process

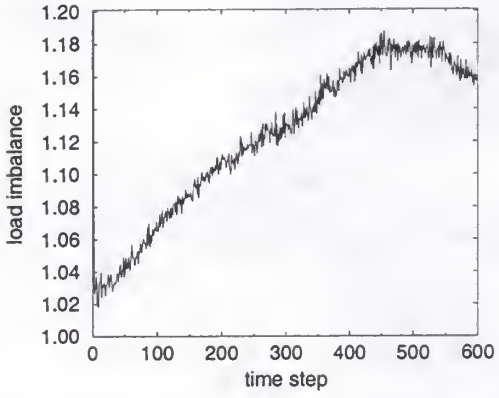


Figure 8.10: History of grid assembly load imbalance based on execution time of the stencil search

of the flow solution. This is the time available to hide the grid assembly time; thus, the grid assembly time is not completely hidden. There is a lot of noise in the flow solver execution time; however, the value is relatively constant. The "idle" time was measured around the synchronization points; thus, this time includes both the time to perform the communications and the time waiting for all of the processes to reach the barrier. Some runs where made with additional synchronization points used to separate the communication times from the wait time. Most of the idle time shown is wait time due to the load imbalance.

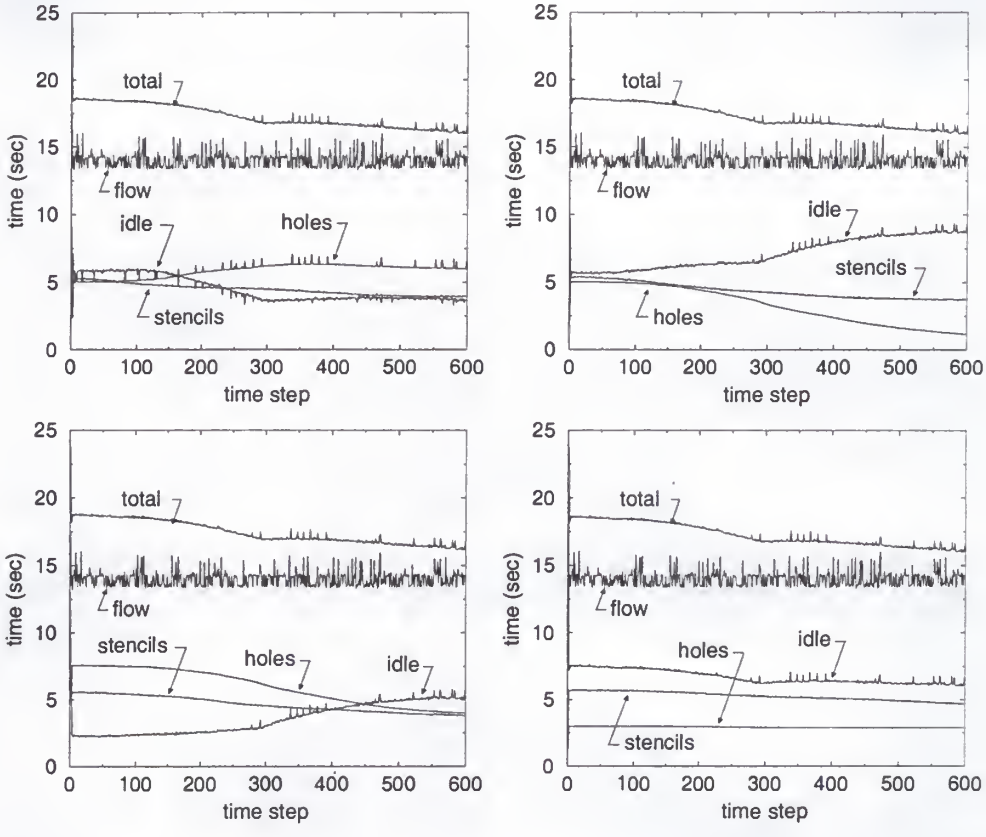


Figure 8.11: Grid assembly execution timings for four FE processes

To better judge the load balance of the hole cutting and stencil search routines, the execution times of these two routines are plotted separately in figure 8.12. The left

plot shows the execution time of hole cutting and the right plot shows the execution time of the stencil search. Each curve in the two plots represents a different FE process. The grouping or separation between the curves represents the variation in the execution times on the different processes; thus, it indicates the quality of the load balance. From these plots, it can be seen that the stencil search routine is much better balanced than the hole cutting routine. This should be the case, since the execution time of the stencil search routine was used to judge the load balance (and used to drive the dynamic load balancing routine). The stencil search imbalance is always less than 10%, while the hole cutting imbalance is in the range of 50%–70%.

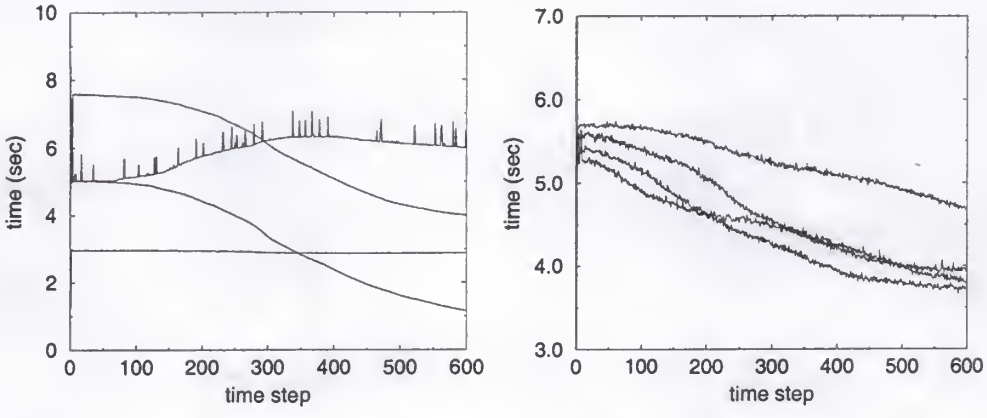


Figure 8.12: Grid assembly execution timings of (left) hole cutting and (right) stencil searching with load balance based on measured execution time of stencil searching. Each curve represents a separate process.

Since system calls are being used to measure the execution times needed to judge the load balance, the time required to obtain the timing information contributes to the processor idle time. Thus, it would be beneficial to define a metric by which the load balance could be judged without introducing additional function calls. The number of IGBPs per process has been used in other references (see Barszcz et al. [36] for example) to judge the load balance. Therefore, some runs where made using the number of IGBPs per superblock as a measure of the grid assembly work associated with the superblock. Figure 8.13 shows the perceived imbalance. From this plot,

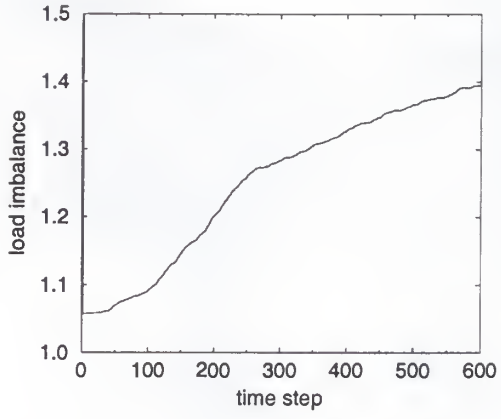


Figure 8.13: History of grid assembly load imbalance based on number of IGBPs

the grid assembly is not well balanced, and the overall performance of the code was significantly worse.

Figure 8.14 shows the execution times for the hole cutting and stencil search routines when the load balance was judged based on the number of IGBPs per superblock. Again, each curve represents execution time on a different FE process. The stencil search routine is still relatively well balanced, but the imbalance in hole cutting has increased significantly. The imbalance in the stencil search is less than 20%, while the imbalance in hole cutting is as much as 160%. This indicates that the actual execution times of the stencil search routines does a much better job at load balancing the work load than does the number of IGBPs.

Table 8.3 summarizes the results from some of the phase III runs. These data represent the position of the CG and angular orientation of the inboard store after the complete trajectory. The maximum differences, as compared to the sequential run, are on the order of 0.001 feet and 0.03 degrees. The use of multiple FE processes to compute the grid assembly function has no effect on the accuracy of the trajectory computed.

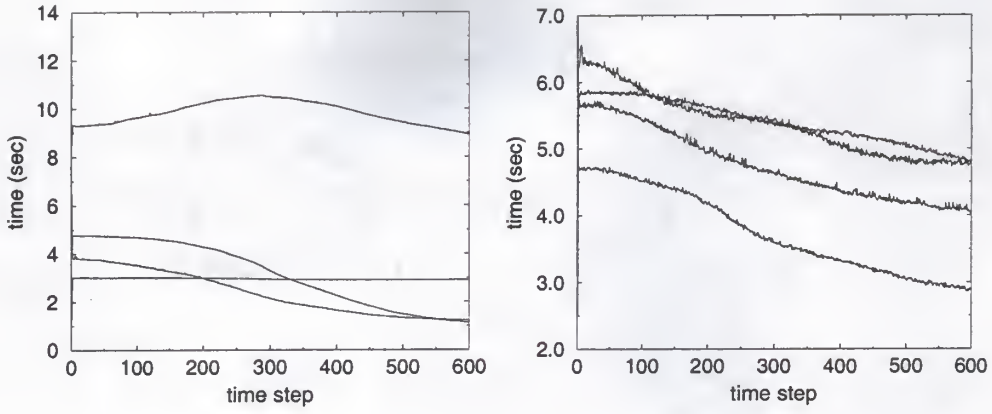


Figure 8.14: Grid assembly execution timings of (left) hole cutting and (right) stencil searching with load balance based on number of IGBPs. Each curve represents a separate process.

Table 8.3: Summary of results from the phase III runs including the final position of the inboard store

		Seq	Phase III				
no. of processors		1	16+4	24+4	28+4	32+4	40+4
position	x	-0.9935	-0.9935	-0.9935	-0.9935	-0.9935	-0.9935
	y	-2.9851	-2.9853	-2.9853	-2.9853	-2.9854	-2.9853
	z	2.0075	2.0064	2.0064	2.0064	2.0064	2.0064
orientation	yaw	16.435	16.448	16.450	16.450	16.450	16.450
	pitch	-18.629	-18.641	-18.641	-18.641	-18.641	-18.641
	roll	-4.6459	-4.6152	-4.6172	-4.6172	-4.6151	-4.6172
exec. time (min)		9384	557	388	390	350	324

Phase IV: Fine Grain Decomposition

The use of superblocks, as a basis for domain decomposition of the grid assembly function, performed relatively well for the ripple release test problem. With only 10 superblocks, it is fortunate that the grid assembly work load balanced relatively well on 4 FE processes. However, the grid assembly time failed to be completely hidden by the execution time of the flow solver when more than 24 BE processes were used. To be able to utilize more FE processes, a fine grain decomposition of the grid assembly work is required.

As was stated in the previous section, the two most expensive routines in the grid assembly function are the hole cutting and the stencil search routines. The work of the hole cutting routine is associated with the number of facets doing the hole cutting, while the work of the stencil search routines is associated with the number of IGBPs that require interpolation. As a first step, the hole cutting routine was decomposed separate from the stencil search routine. The number of hole cutting facets was used as the basis for fine grain decomposition of the hole cutting routine, while the stencil search routine and the remainder of the grid assembly function were still decomposed based on superblocks. Since the hole cutting is now decoupled from the distribution of the superblocks, the redistribution of superblocks for dynamic load balancing is based on the execution times of the stencil search routine.

As an initial step, the total number of hole cutting facets is equally divided between the FE processes. Each FE process cuts holes into all of the superblocks with all of the facets that have been mapped to that process. The cell state information is stored in shared memory to avoid an expensive reduction operation that would be required to merge the cell state information after the holes are outlined.

There are two options for cutting holes. As mentioned in chapter 2, the default option outlines the holes and then fills them with a fast sweep through the grids. The "nofill" option outlines the holes but does not fill them. With the "nofill" option, less work has to be done when outlining the holes, because the facets do not have to be refined to ensure a complete outline. However, more work has to be done when searching for interpolation stencils, because the points which are actually inside a hole will fail interpolation. For the sequential run of the ripple release problem, the use of either option does not make a significant difference in the execution times. Therefore, all of the runs up to this point (including the sequential run) were done using the "nofill" option. However, the new implementation should make it easier to use more processes to reduce the execution time of hole cutting; therefore, the best performance should be seen if some of the work of the stencil search can be shifted to

the hole cutting function. Thus, the "nofill" option is not used. The default option of outlining and filling the holes is used.

The ripple release test problem was run with the fine grain hole cutting on 4 FE processes and 28–40 BE processes. The timing results are displayed in figure 8.15 along with the previous results from the phase III runs. The grid assembly time is now completely hidden by the execution time of the flow solver for the run with 28 BE processes. In fact, the 28+4 process run outperforms the phase III run on 40+4 processors. The 32+4 and 36+4 runs are in the region of the "bend" in the performance curve and are probably affected by the use of average work fractions to predict the performance. However, they actually performed slightly worse than the 28+4 run. Therefore, they may have been affected by uncontrollable machine load or other conditions. The 40+4 process run performed quite well. Solving for the grid assembly imbalance using equations 6.4 and 6.7 and the speedup from the 40+4 process run, the imbalance is 38% as compared to the 65% for the phase III run.

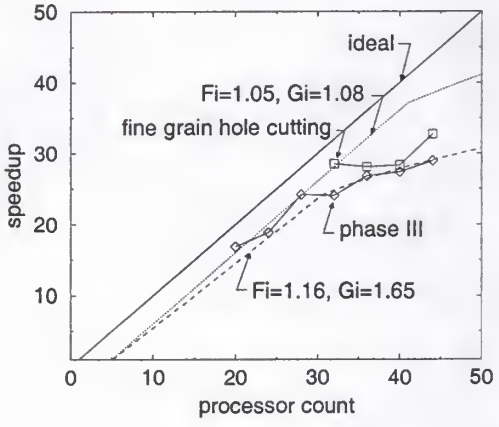


Figure 8.15: Speedup due to fine grain hole cutting and load balancing of hole cutting separate from the stencil search

The execution of the stencil search routine should be equivalent to that in the phase III runs. The improvements in speedup, seen in figure 8.15, are due to the improvement in the load balancing of the hole cutting. Figure 8.16 shows the execution

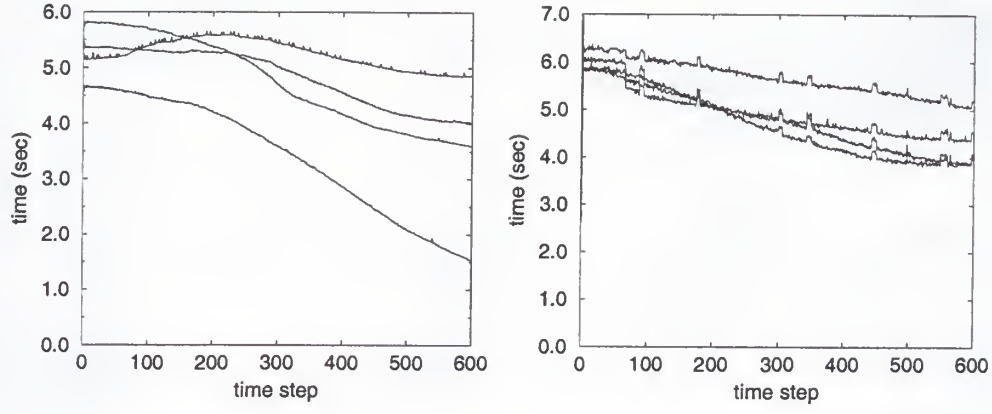


Figure 8.16: Grid assembly execution timings of (left) hole cutting and (right) stencil searching with fine grain hole cutting and the stencil search load balanced based on execution time. Each curve represents a separate process.

times of the hole cutting routine and the stencil search routines throughout the 40+4 process run. Each curve represents the execution time on a different FE process. These plots should be comparable to the plots in figure 8.12, although the total hole cutting time has increased and the total stencil search time has decreased, because of the use of the "nofill" option in the previous runs. Overall, the load balance of the hole cutting has improved. However, as the computation progresses, the execution times of the hole cutting on the different processes tends to spread apart. Thus, the load imbalance is increasing as the stores move apart.

In order to demonstrate the advantage of the fine grain decomposition in using more FE processes, runs were made with 40 BE processes and 5, 6, 7, and 8 FE processes. The execution times for the hole cutting and stencil search routines are shown in figures 8.17–8.20 for these runs. Each set of plots contains 5, 6, 7, and 8 curves, respectively, representing the execution times on the different FE processes. In general, the execution times of the hole cutting routine decreases with the additional FE processes, although some significant load imbalances are seen. This is indicated by the progressive decrease in the average execution times of the hole cutting on the different FE processes. The good load balance is indicated by the tight grouping of

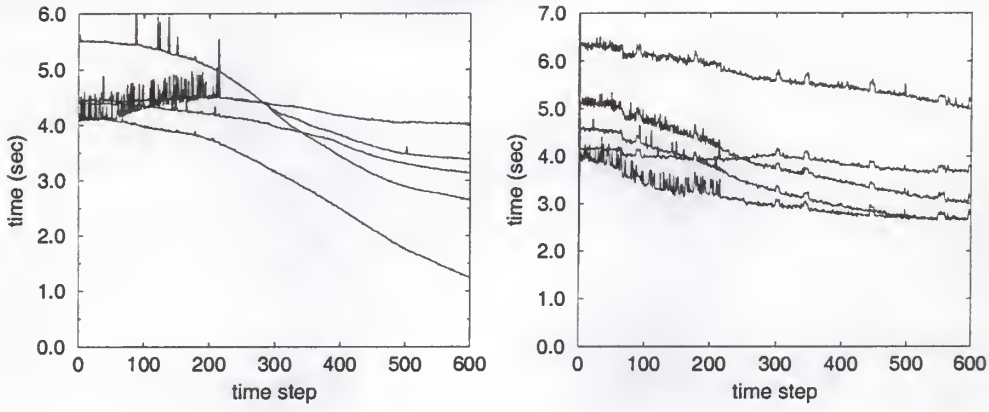


Figure 8.17: Grid assembly execution timings of (left) hole cutting and (right) stencil searching with fine grain hole cutting and the stencil search distributed across 5 FE processes. Each curve represents a separate process.

the curves in the figures; however, some of the execution times deviate significantly from the average, indicating a load imbalance.

Since there are 10 superblocks, we can continue to distribute the superblocks over the FE processes. However, there is one superblock that dominates the stencil search execution time. This superblock is composed of a set of grids in the region of the pylon and the stores. It is used as a interface between the store grids and the wing grids and is used to improve the resolution of the flow in the region of the trajectory of the stores. All of the stores cut holes into this superblock, and it provides the source for many interpolations. The total work associated with this superblock is about 1/4 of the total grid assembly work; therefore, it does not adversely affect the load balance on the 4 FE process runs. However, as more FE processes are used, the work associated with this superblock can not be subdivided and the total execution time of the stencil search does not decrease (similar to the situation discussed relative to figure 5.2). This can be seen as the execution time of the stencil search for one of the processes always starts just above 6 seconds and finishes just below 5 seconds. Remember, the total execution time of a function distributed over multiple processes is dictated by the maximum execution time of all of the processes.

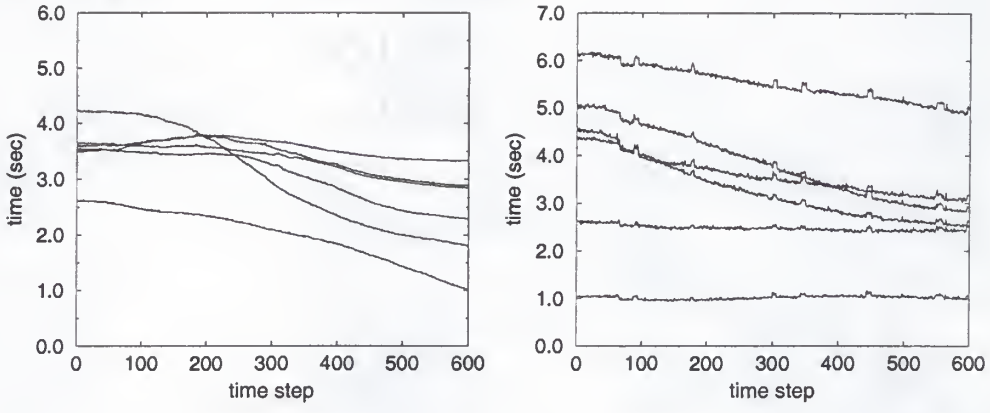


Figure 8.18: Grid assembly execution timings of (left) hole cutting and (right) stencil searching with fine grain hole cutting and the stencil search distributed across 6 FE processes. Each curve represents a separate process.

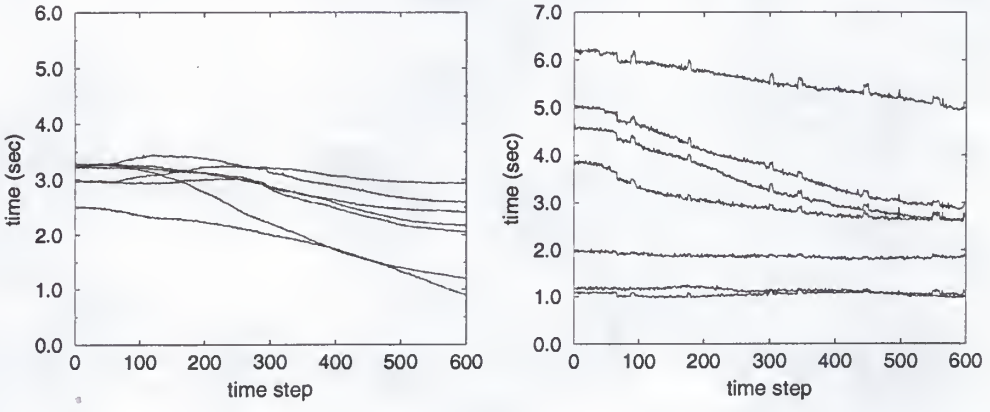


Figure 8.19: Grid assembly execution timings of (left) hole cutting and (right) stencil searching with fine grain hole cutting and the stencil search distributed across 7 FE processes. Each curve represents a separate process.

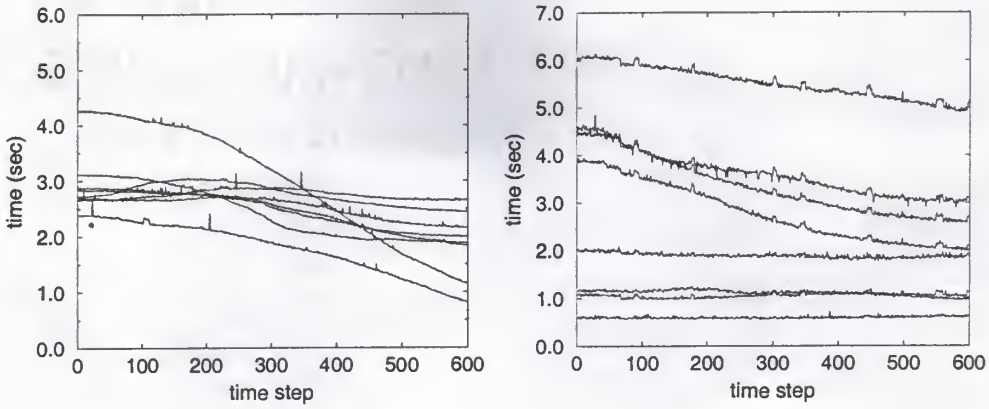


Figure 8.20: Grid assembly execution timings of (left) hole cutting and (right) stencil searching with fine grain hole cutting and the stencil search distributed across 8 FE processes. Each curve represents a separate process.

Figure 8.21 shows the timing results from these runs. The data with the square symbols are for the runs with 4 FE processes and varying numbers of BE processes. The data with the circle symbols are the runs with 40 BE processes and 5–8 FE processes. The 5 FE process execution histories shown in figure 8.17 show some excessive noise, which might indicate some uncontrollable machine load or other condition. This might account for the dropoff in performance on the 5 FE process run. The 8 FE process also showed a dropoff in performance. This is most likely due to the large load imbalance seen in the hole cutting for this case.

The failure to maintain a good load balance in the hole cutting is due to the fact that no dynamic load balancing is being employed. The total number of hole cutting facets is equally divided among the available FE processes. However, each facet does not do an equal amount of work. Some facets will overlap only one grid, while other facets overlap many grids. Likewise, some facets may overlap a region of tightly spaced grid cells and will require refinement, while other facets do not require refinement.

In order to use the same dynamic load balancing algorithm that is used to redistribute the superblocks, the execution time of each hole cutting facet would have

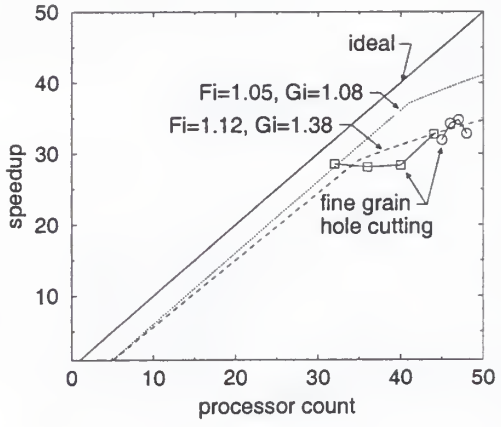


Figure 8.21: Use of additional processors continues to reduce time for hole cutting

to be measured separately. Since there are about 60,000 hole cutting facets in the ripple release test problem, measuring the execution time of each facet could add a significant amount of system time. Likewise, the load balancing routine used to redistribute the superblocks is not optimized to handle the sorting and searching required by that many individual pieces of work. Therefore, the load balancing of the fine grain hole cutting is based on an algorithm that uses the execution time of the hole cutting routine per FE process as a weight. This algorithm was described in chapter 5. The algorithm varies the number of facets assigned to each FE process so that the weighted number of facets is evenly distributed.

Figures 8.22–8.26 show the execution times of the hole cutting routine when 4–8 FE processes are used in combination with dynamic load balancing of the hole cutting. These figures may appear to contain only a single curve; however, each plot actually contains from 4 to 8 curves representing the execution time for hole cutting on the different FE processes. The tight grouping of the curves indicates the remarkable load balance that was achieved. It takes several time steps at the beginning of a run for the execution times to converge, and small changes in the execution of the computer, which can cause variations in the execution time of a process, can cause perturbations in the load balance. The load balance quickly recovers from these; however, some sort

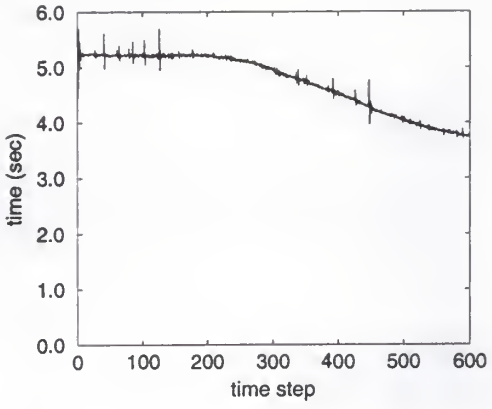


Figure 8.22: Execution times for load balanced fine grain hole cutting distributed across 4 FE processes

of limiting or damping of the changes between the time steps may help to maintain the load balance.

Unfortunately, with limited, shared computing resources, the runs using fine grain hole cutting with dynamic load balancing could not be repeated for overall speedup measurements. The timings presented in figures 8.22–8.26 were taken while running the code in a special mode to allow prescribed motion. The flow solution was not calculated. Instead, the motion was prescribed from a recording of the motion calculated on previous runs. The work of the grid assembly is the same as if the motion had been driven by the flow solution, because all of the grids are placed in the appropriate positions throughout the trajectory. However, computing resources are minimized because only the FE processes are needed.

Table 8.4 summarizes the results from some of the phase IV runs in which the motion was driven by the flow solution. These data represent the position of the CG and angular orientation of the bottom store after the complete trajectory. The maximum differences, as compared to the sequential run, are on the order of 0.005 feet and 0.1 degrees. The use of additional FE processes and fine grain hole cutting has no affect on the accuracy of the trajectory computed.

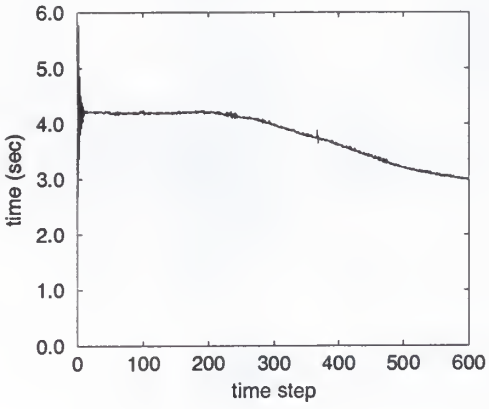


Figure 8.23: Execution times for load balanced fine grain hole cutting distributed across 5 FE processes

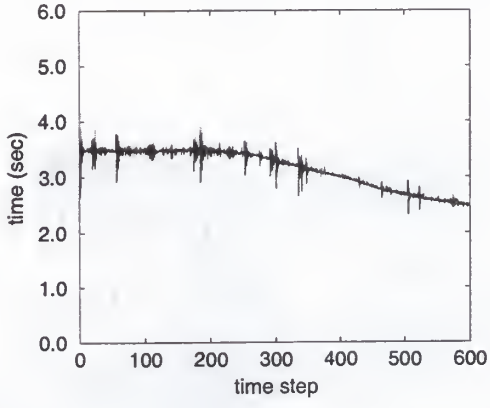


Figure 8.24: Execution times for load balanced fine grain hole cutting distributed across 6 FE processes

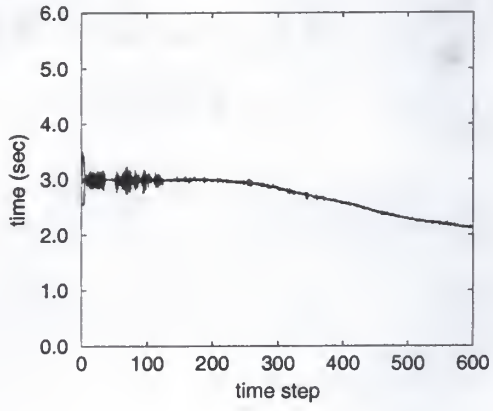


Figure 8.25: Execution times for load balanced fine grain hole cutting distributed across 7 FE processes

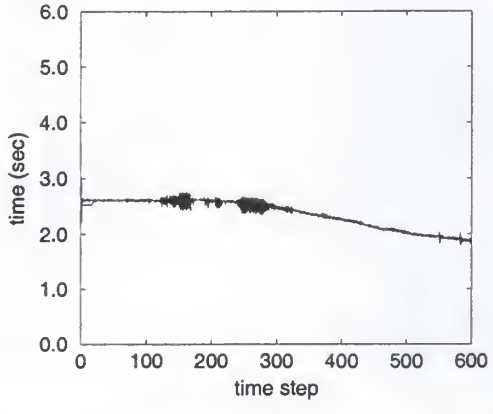


Figure 8.26: Execution times for load balanced fine grain hole cutting distributed across 8 FE processes

Table 8.4: Summary of results from the runs that used fine grain hole cutting including the final position of the bottom store

		Seq	Phase IV			
no. of processors		1	28+4	32+4	36+4	40+4
position	x	-1.5868	-1.5883	-1.5883	-1.5883	-1.5883
	y	-0.0010	0.0010	0.0010	0.0010	0.0010
	z	6.5311	6.5268	6.5268	6.5268	6.5268
orientation	yaw	8.0537	8.0579	8.0579	8.0579	8.0579
	pitch	3.4349	3.3067	3.3069	3.3065	3.3069
	roll	1.3667	1.3878	1.3878	1.3878	1.3877
exec. time (min)		9384	329	334	331	287

Summary

Table 8.5 gives a listing of some of the best execution times from runs of the ripple release test problem using the different implementations. The number of BE processes is listed in the first column and the number of FE processes is listed in parentheses beside the execution times. All of the phase I runs used a single FE process to perform the complete grid assembly in a serial fashion with respect to the parallel execution of the flow solver. The sequential run time is listed in the first row under phase I, although there was only one process containing both the flow solver and the grid assembly functions. The phase II timings show the improvement gained by overlapping the grid assembly execution time and the flow solver execution time. The phase III timings show the continued improvement due to coarse grain decomposition of the grid assembly function based on the distribution of the superblocks. The phase IV timings show the improvement due to the distribution of the work of the stencil search and the hole cutting functions using separate bases. The stencil search is still based on coarse grain decomposition based on superblocks, while the hole cutting is based on fine grain decomposition based on the hole cutting facets (without dynamic load balancing for these timings). The best timing is for 40 BE processes and 7 FE processes using the fine grain hole cutting. The total execution time was decreased from about 6.5 days to 4.5 hours.

Table 8.5: Summary of best execution times (in minutes) from runs of the different implementations (number of FE processes shown in parentheses)

	Phases			
NBES	I	II	III	IV
1	9384 (1)	-	-	-
4	2786 (1)	1920 (1)	-	-
8	1543 (1)	1041 (1)	-	-
12	1150 (1)	920 (1)	-	-
16	1013 (1)	729 (1)	557 (4)	-
20	-	-	498 (4)	-
24	-	-	388 (4)	-
28	-	-	390 (4)	329 (4)
32	-	-	350 (4)	334 (4)
36	-	-	343 (4)	331 (4)
40	-	-	324 (4)	270 (7)

Figure 8.27 is a combination of figures 8.1, 8.5, 8.7 and 8.15, showing speedup from most of the data listed in table 8.5. Some estimated speedup curves are also included to show trends in the data and the performance that can be expected from the use of different numbers of processors. The four curves and associated data are from the four phases of implementation. The data labeled "Fine grain GA" are from the phase IV runs with fine grain hole cutting without dynamic load balancing of the hole cutting. Overall, the performance has increased with each successive implementation.

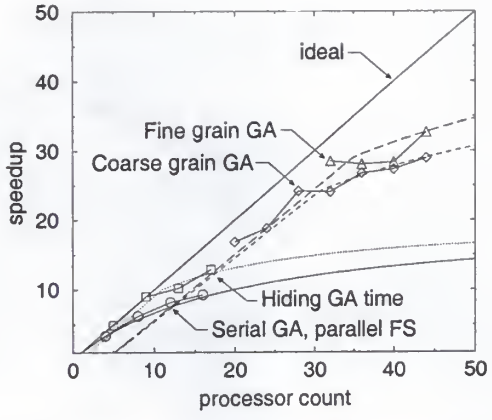


Figure 8.27: Summary of the increasing speedup achieved through the different implementations

CHAPTER 9
CONCLUSIONS AND FUTURE WORK

The numerical solution of store separation trajectories from first principles is a computationally expensive task. The use of overlapping, Chimera grids eases the grid generation burden, makes the flow solution process more efficient, and allows for grid movement. However, each time a grid is moved, the communication links between the overlapping grids must be reestablished. This makes the numerical solution of store separation trajectories even more costly. Therefore, it is important to address the use of parallel processing to reduce the computation time required to calculate store separation trajectories. Paramount in this effort is the requirement to parallelize the execution of the grid assembly function.

The parallel implementation of the grid assembly function was addressed and four implementations were presented. The performance of each implementation was analyzed and the weaknesses were attacked, with each successive implementation, in an effort to improve performance. The first implementation took the easiest approach to solving dynamic, moving body problems in parallel. One process was used for grid assembly, while multiple processes were used for the flow solver. The execution time of the grid assembly function was not decreased; however, multiple processes were used to decrease the execution time of the flow solver. Thus, the wall clock time needed to compute store separation trajectories was decreased by using parallel computing.

In the second implementation, the parallel performance was improved by reducing the serial fraction of the work. This was done by hiding some or all of the execution time of the grid assembly function behind the execution time of the flow solver. This technique is similar to approaches used in serial implementations, but

this is the first time that this technique has been used in the parallel implementation of Chimera grid schemes. The opportunity to hide the grid assembly time arises from the Newton relaxation scheme used in the flow solver. Therefore, this implementation also emphasizes the need to consider the entire calculation to achieve better performance.

The third implementation used multiple processes to decrease the execution time of the grid assembly function. This makes it easier to hide the execution time of the grid assembly function behind the execution time of the flow solver. A coarse grain data decomposition of the grid assembly function was used based on superblocks. The superblocks were distributed across the available processes. On each process, holes were cut into and interpolation sources were identified only for the superblocks mapped to that process. The work load associated with each superblock was measured by the execution time and a dynamic load balancing algorithm was devised to redistribute the superblocks in order to achieve a good load balance. This represents the first time that a grid assembly function has been parallelized and dynamic load balancing was used based on a decomposition that is separate from that of the flow solver.

The relatively small number of superblocks in the test problem placed a limit on the number of processes that could be used effectively to decrease the execution time of the grid assembly function. Therefore, the final implementation demonstrated the use of fine grain data decomposition of the work associated with grid assembly to improve scalability. In this implementation, the hole cutting facets were used as the basis for the data decomposition of the work associated with the hole cutting portion of the grid assembly function. The hole cutting facets were distributed across the available processes, and each process cut holes into all of the grids using the facets that were mapped to that process. Each facet cuts holes independent of the other facets; however, the resulting cell state information, used to track the holes, must be complete for each grid. Therefore, shared memory was used to store the cell state

information, and an expensive reduction operation, needed to recombine the cell state information (if it had been distributed across multiple processes), was avoided.

The work associated with each facet is not uniform; therefore, a dynamic load balancing algorithm was devised based on the use of the execution time of a process as a weight to the cost of the facets mapped to that process. This algorithm proved to be effective at maintaining a near optimum load balance throughout the test problem calculation shown. The best performance seen in the calculations presented reduced the execution time of the test problem from 6.5 days on a single process to 4.5 hours on 47 processes, representing a 34.8 times reduction in the wall clock execution time.

The distribution of the IGBPs across multiple processes would allow for a fine grain decomposition of the remaining work of the grid assembly function. This would allow for the scalable execution of the grid assembly function on larger numbers of processes. With the significant reduction in the execution time of the grid assembly function, the grid assembly would be performed after the final forces and moments are computed and there would be no question about errors introduced into the trajectory. All of the available processes could be used for both the flow solver and the grid assembly. In order to do this, the PM tree must be stored in shared memory so that all of the processes can access it during grid assembly.

The use of shared memory limits the computing resources that can be effectively utilized. An alternative method that can effectively use distributed memory machines should be devised. One method is to use a system level library that mimics shared memory on distributed memory machines. Several groups have worked on this functionality but there are no production implementations currently available and the performance is not known. Alternatively, with PVM's abilities for heterogeneous computing environments, the use of shared memory programming in combination with message passing can allow clusters of shared memory machines to be used as one computing resource.

Another item that should be addressed is the decomposition and load balancing

of the flow solver. The current method of splitting grids to create smaller pieces of work that will allow better load balancing is cumbersome. The grids must be split, flow solutions must be copied from previous grids, and the final solutions reflect the split grid system which complicates visualization. This process should be automated so that the user does not have to be involved in the splitting of grids and the visualization tasks can be performed with the original grids. However, as more and more processes are used, the splitting of grids reduces the implicit nature of the solution and can adversely affect the solution convergence. Shared memory techniques should be investigated to decompose the work of the flow solver without splitting the grids or affecting the solution.

BIBLIOGRAPHY

[1] Benek, J.A., Steger, J.L., and Dougherty, F.C., "A Flexible Grid Embedding Technique with Applications to the Euler Equations," AIAA Paper 83-1944, June 1983.

[2] Meakin, R.L., "A New Method for Establishing Intergrid Communication among Systems of Overset Grids," AIAA Paper 91-1586, June 1991.

[3] Maple, R.C., and Belk, D.M., "Automated Set Up of Blocked, Patched, and Embedded Grids in the Beggar Flow Solver," *Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, ed. N.P. Weatherill et al., Pine Ridge Press, pp. 305-314, 1994.

[4] Prewitt, N.C., Belk, D.M., and Maple, R.C., "Multiple Body Trajectory Calculations Using the Beggar Code," AIAA Paper 96-3384, July 1996.

[5] Belk, D.M., and Maple, R.C., "Automated Assembly of Structured Grids For Moving Body Problems," AIAA Paper 95-1680, June 1995.

[6] Belk, D.M., and Strasburg, D.W., "Parallel Flow Solution on the T3D with Blocked and Overset Grids," 3rd Symposium on Overset Composite Grid and Solution Technology, November 1996.

[7] Roe, P.L., "Approximate Riemann Solvers, Parameter Vectors and Difference Schemes," *Journal of Computational Physics*, Vol. 43, No. 2, pp. 357-372, October 1981.

[8] Whitfield, D.L., "Newton-Relaxation Schemes for Nonlinear Hyperbolic Systems," Engineering and Industrial Research Station Report MSSU-EIRS-ASE-90-3, Mississippi State University, October 1990.

[9] Suhs, N.E., and Tramel, R.W., "PEGSUS 4.0 USER'S MANUAL," AEDC-TR-91-8, June 1991.

[10] Buning, P.G., and Chan, W.M., "OVERFLOW/F3D User's Manual, Version 1.5," NASA/ARC, November 1990.

[11] Chiu, Ing-Tsau, and Meakin, R.L, "On Automating Domain Connectivity for Overset Grids," NASA-CR-199522, August 1995.

[12] Chesshire, G., and Henshaw, W.D., "Composite Overlapping Meshes for the Solution of Partial Differential Equations," *Journal of Computational Physics*, Vol. 90, No. 1, September 1990.

[13] Chesshire, G., Brislawn, D., Brown, D., Henshaw, W., Quinlan, D., and Saltzman, J., "Efficient Computation of Overlap for Interpolation between Moving Component Grids," 3rd Symposium on Overset Composite Grid and Solution Technology, November 1996.

[14] Dillenius, M., Lesieutre, D., Whittaker, C., and Lesieutre, T., "New Applications of Engineering Level Missile Aerodynamics and Store Separation Prediction Methods," AIAA Paper 94-0028, January 1994.

[15] Carman, J.B., Jr., "Store Separation Testing Techniques at the Arnold Engineering Development Center - Volume 1 - An Overview," AEDC-TR-79-1 (AD-A088583), Vol. 1, August 1980.

[16] Keen, K.S., "New Approaches to Computational Aircraft/Store Weapons Integration," AIAA Paper 90-0274, January 1990.

[17] Jordan, J.K., "Computational Investigation of Predicted Store Loads in Mutual Interference Flow Fields," AIAA Paper 92-4570, August 1992.

[18] Witzeman, F., Strang, W., and Tomaro, R., "A Solution on the F-18C for Store Separation Simulation using COBALT," AIAA Paper 99-0122, January 1999.

[19] Bruner, C., and Woodson, S., "Analysis of Unstructured CFD Codes for the Accurate Prediction of A/C Store Trajectories," AIAA Paper 99-0123, January 1999.

[20] Welterlen, T., "Store Release Simulation on the F/A-18C Using Split Flow," AIAA Paper 99-0124, January 1999.

[21] Benmeddour, A., Fortin, F., and Jones, D., "Application of the Canadian Code to the F/A-18C JDAM Separation," AIAA Paper 99-0127, January 1999.

[22] Bayyuk, S.A., Powell, K.G., and van Leer, B., "A Simulation Technique for 2-D Unsteady Inviscid Flows Around Arbitrarily Moving and Deforming Bodies of Arbitrary Geometry," AIAA Paper 93-3391, July 1993.

[23] Arabshahi, A., and Whitfield, D.L., "A Multiblock Approach to Solving the Three-Dimensional Unsteady Euler Equations about a Wing-Pylon-Store Configuration," AIAA Paper 89-3401, August 1989.

[24] Singh, K.P., Newman, J.C., and Baysal, O., "Dynamic Unstructured Method for Flows Past Multiple Objects in Relative Motion," *AIAA Journal*, Vol. 33, No. 4, pp. 641-649, April 1995.

[25] Karman, S.L., "SPLITFLOW: A 3D Unstructured Cartesian/Prismatic Grid CFD Code for Complex Geometries," AIAA Paper 95-0343, January 1995.

[26] Yen, G., and Baysal, O., "Dynamic-Overlapped-Grid Simulation of Aerodynamically Determined Relative Motion," AIAA Paper 93-3018, July 1993.

[27] Blaylock, T.A., "Application of the FAME method to the Simulation of Store Separation from a Combat Aircraft at Transonic Speed," Numerical Grid Generation in Computational Field Simulations; Proceedings of the 5th International Conference, pp. 805-814, April 1996.

[28] Lijewski, L.E., and Suhs, N.E., "Chimera-Eagle Store Separation," AIAA Paper 92-4569, August 1992.

[29] Lijewski, L.E., and Suhs, N., "Time-Accurate Computational Fluid Dynamics Approach to Transonic Store Separation Trajectory Prediction," *Journal of Aircraft*, Vol. 31, No. 4, pp. 886-891, August 1994.

[30] Thoms, R.D., and Jordan, J.K., "Investigations of Multiple Body Trajectory Prediction Using Time Accurate Computational Fluid Dynamics," AIAA Paper 95-1870, June 1995.

[31] Cline, D.M., Riner, W., Jolly, B., and Lawrence, W., "Calculation of Generic Store Separations from an F-16 Aircraft," AIAA Paper 96-2455, June 1996.

[32] Coleman, L., Jolly, B., Chesser, B.L., Jr., and Brock, J.M., Jr., "Numerical Simulation of a Store Separation Event from an F-15E Aircraft," AIAA Paper 96-3385, July 1996.

[33] Smith, M.H., and Pallis, J.M., "MEDUSA - An Overset Grid Flow Solver for Network-based Parallel Computer Systems," AIAA Paper 93-3312, July 1993.

[34] Wissink, A.M., and Meakin, R.L., "Computational Fluid Dynamics with Adaptive Overset Grids on Parallel and Distributed Computer Platforms," International Conference on Parallel and Distributed Computing, July 1998.

[35] Meakin, R.L., and Wissink, A.M., "Unsteady Aerodynamic Simulation of Static and Moving Bodies Using Scalable Computers," AIAA Paper 99-3302, July 1999.

[36] Barszcz, E., Weeratunga, S.K., and Meakin, R.L., "Dynamic Overset Grid Communication on Distributed Memory Parallel Processors," AIAA Paper 93-3311, July 1993.

[37] Weeratunga, S.K., and Chawla, K., "Overset Grid Applications on Distributed Memory MIMD Computers," AIAA Paper 95-0573, January 1995.

[38] Wissink, A.M., and Meakin, R.L., "On Parallel Implementations of Dynamic Overset Grid Methods," SC97: High Performance Networking and Computing, November 1997.

[39] Prewitt, N.C., Belk, D.M., and Shyy, Wei, "Implementations of Parallel Grid Assembly for Moving Body Problems," AIAA Paper 98-4344, August 1998.

[40] Prewitt, N.C., Belk, D.M., and Shyy, Wei, "Distribution of Work and Data for Parallel Grid Assembly," AIAA Paper 99-0913, January 1999.

[41] Samet, H., *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA, 1990.

[42] Samet, H., *Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990.

[43] Potter, M.C., and Foss, J.F., *Fluid Mechanics*, Great Lakes Press, Okemos, MI, 1982.

[44] Belk, D.M. *Unsteady Three-Dimensional Euler Equations Solutions on Dynamic Blocked Grids*, PhD Dissertation, Mississippi State University, August 1986.

[45] Steger, J.L., and Warming, R.F., "Flux Vector Splitting of the Inviscid Gas-Dynamic Equations with Application to Finite Difference Methods," *Journal of Computational Physics*, Vol. 40, No. 2, pp. 263-293, April 1981.

[46] Hirsch, C., *Numerical Computation of Internal and External Flows - Volume 2: Computational Methods for Inviscid and Viscous Flow*, John Wiley & Sons, Chichester, England, 1990.

[47] Briley, W.R., and McDonald, H., "Solution of the Multi-Dimensional Compressible Navier-Stokes Equations by a Generalized Implicit Method," *Journal of Computational Physics*, Vol. 24, pp. 372-397, 1977.

[48] Beam, R.M., and Warming, R.F., "An Implicit Finite-Difference Algorithm for Hyperbolic Systems in Conservation Law Form," *Journal of Computational Physics*, Vol. 22, pp. 87-110, 1976.

[49] Van Leer, B., "Towards the Ultimate Conservation Difference Scheme. V. A Second Order Sequel to Godunov's Method," *Journal of Computational Physics*, Vol. 32, No. 1, pp. 101-136, July 1979.

[50] Godunov, S.K., "A Difference Scheme for Numerical Computation of Discountinuous Solution of Hydrodynamic Equations," *Math. Sbornik*, Vol. 47, pp. 271-306, 1959 (in Russian). Translated US Joint Publ. Res. Service, JPRS 7226, 1969.

[51] Whitfield, D., and Taylor, L., "Discretized Newton-Relaxation Solution of High Resolution Flux-Difference Split Schemes," AIAA Paper 91-1539, June 1991.

[52] Rizk, M.H., "The Use of Finite-Differenced Jacobians for Solving the Euler Equations and for Evaluating Sensitivity Derivatives," AIAA Paper 94-2213, June 1994.

[53] Conte, S.D., and de Boor, C., *Elementary Numerical Analysis - An Algorithmic Approach*, third edition, McGraw-Hill, New York, NY, 1980.

[54] Meakin, R.L., "Computations of the Unsteady Flow About a Generic Wing/Pylon/Finned-Store Configuration," AIAA Paper 92-4568, August 1992.

[55] Etkin, B., *Dynamics of Flight – Stability and Control*, John Wiley & Sons, New York, NY, 1982.

[56] Blakelock, J.H., *Automatic Control of Aircraft and Missiles*, John Wiley & Sons, New York, NY, 1965.

[57] Nash, J.R., "Derivation Using Quaternions in the Simulation of Rigid Body Motion," November 1983.

[58] Hamilton, W.R., "On a New Species of Imaginary Quantities Connected with a Theory of Quaternions," *Proceedings of the Royal Irish Academy, Dublin*, Vol. 2, No. 13, pp. 424-434, 1843.

[59] Robinson, A.C., "On the Use of Quaternions in Simulation of Rigid Body Motion," USAF Wright Air Development Center, TR 58-17, Dayton, OH, December 1958.

[60] Katz, A., "Special Rotation Vectors: A Means for Transmitting Quaternions in Three Components," *AIAA Journal of Aircraft*, Vol. 30, No.1, January 1993.

[61] Foster, Ian, *Designing and Building Parallel Programs*, Addison-Wesley, Reading, MA, 1995.

[62] Sterling, T.L., Salmon, J., Becker, D.J., and Savarese, D.F., *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*, MIT Press, Cambridge, MA, 1999.

[63] Jiang, D., and Singh, J.P., "A Scaling Study of the SGI Origin2000: A Hardware Cache-Coherent Multiprocessor," Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing, March 1999.

[64] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V., *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, 1994.

[65] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," *International Journal of Supercomputing Applications*, Vol. 8, No. 3/4, 1994.

[66] McBryan, O.A., "An Overview of Message Passing Environments," *Parallel Computing*, Vol. 20, pp. 417-444, 1994.

[67] Nichols, B., Buttlar, D., and Farrel, J.P., *Pthreads Programming*, O'Reilly & Associates, Sebastopol, CA 1996.

[68] Dagum, L., and Menon, R., "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, Vol. 5, No. 1, January/March 1998.

[69] Gallmeister, B.O., *POSIX.4: Programming for the Real World*, O'Reilly & Associates, Sebastopol, CA, 1995.

[70] Blosch, E.L., and Shyy, W., "Scalability and Performance of Data-Parallel Pressure-Based Multigrid Methods for Viscous Flows," Journal of Computational Physics, Vol. 125, No. 2, pp. 338-353, May 1996.

[71] Roose, D., and Van Driessche, R., "Parallel Computers and Parallel Algorithms for CFD: An Introduction," AGARD Report R-807, pp. 1-1–1-23, October 1995.

[72] Amdahl, G., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in AFIPS Conference proceedings, Vol. 30, Altlantic City, NJ, pp. 483-485, 1967.

[73] Sedgewick, R., *Algorithms in C++*, Addison-Wesley, Reading, MA, pp. 596-598, 1992.

[74] Dewdney, A.K., *The New Turing Omnibus: 66 Excursions in Computer Science*, Computer Science Press, New York, NY, pp. 210-206, 1993.

BIOGRAPHICAL SKETCH

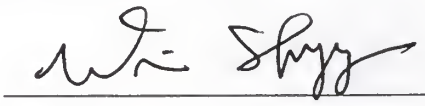
Nathan Coleman Prewitt was born on worldwide communion Sunday, October 4, 1964, in Batesville, Mississippi. He is the youngest of seven children of Maston Levi Prewitt, a Methodist preacher, and China Ray Prewitt, a devoted wife and mother. The Coleman name is a family name from his mother's side (China Ray's maiden name is Yeates). His siblings consist of five sisters (one of whom died as an infant) and, fortunately, one brother, two years his senior. As the son of a Methodist preacher, Nathan moved every 3 or 4 years, growing up in small Mississippi towns such as Merigold, Boyle, Jumpertown, Sunflower, Friars Point, Cleveland, Lyon, and Pickens.

Nathan has always been an avid sports fan. However, a few things, such as size, strength, speed and coordination, prevented him from being an athlete. So Nathan played in the band. Nathan played trumpet at Indianola Academy, Cleveland High School, and Lee Academy, where he graduated from high school. Nathan then played for three years in the Famous Maroon Band at Mississippi State University.

After co-op'ing with the Naval Sea Systems Command in Washington, D.C., Nathan graduated summa cum laude with a B.S. in aerospace engineering from Mississippi State University in 1987. Nathan stayed in Starkville to receive his M.S. in aerospace engineering from Mississippi State University in 1988, concentrating on aerodynamics and computational fluid dynamics. After receiving his master's degree, Nathan accepted a job with Sverdrup Technology at Eglin Air Force Base, FL. Nathan has changed employers and is currently an employee of CACI at Eglin Air Force Base.

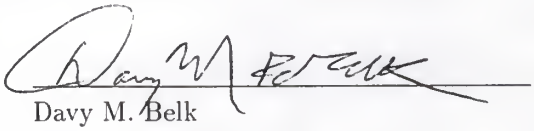
Nathan married Tracye on May 25, 1985. They have two sons: Jacob Coleman, born September 17, 1993, and Joshua Malone, born August 30, 1995 (Malone is Tracye's maiden name). The boys are getting into sports through T-ball and soccer, and they all enjoy returning to Starkville to root for the Dawgs.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



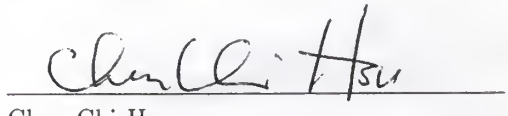
Wei Shyy
Professor of Aerospace Engineering,
Mechanics and Engineering Science

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



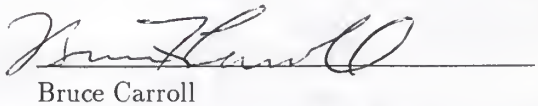
Davy M. Belk
Assistant Professor of Aerospace
Engineering, Mechanics and
Engineering Science

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Chen Chi Hsu
Professor of Aerospace Engineering,
Mechanics and Engineering Science

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



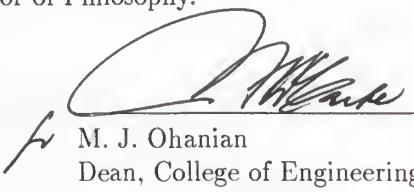
Bruce Carroll
Associate Professor of Aerospace
Engineering, Mechanics and
Engineering Science

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Baba C. Vemuri
Professor of Computer and Information
Science and Engineering

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

December 1999



M. J. Ohanian
Dean, College of Engineering

Winfred M. Phillips
Dean, Graduate School